Department of Electrical and Computer Engineering

University of Thessaly

Master Thesis

Parameterization and Optimization of Neural Networks for Distributed Systems Hadoop-like

---

By

Kotoulas Nikolaos

UNIVERSITY OF THESSALY

DEPARTEMENT OF ELECTRICAL AND COMPUTER

ENGINEERING

Parameterization and Optimization of neural networks for Distributed Systems Hadoop-like

(Παραμετροποίηση και Βελτιστοποήση Νευρωνικών Δικτύων σε Κατανεμημένα Συστήματα τύπου Hadoop)

Master Thesis

Kotoulas Nikolaos

Supervising Professors: Katsaros Dimitrios

Assistant Professor

Stamoulis Georgios

Professor

Tsoukalas Eleutherios

Professor

Contents

Acknowledgements

At this point, I would like to thank all those who have contributed to the completion of this work. First of all, I would like to thank my supervisor Dimitrios Katsaros who inspired me and gave me the opportunity to work in such an interesting project. I am grateful to Certh's system and networks administrator Filimon Georgiou for his valuable knowledge and the help throughout the project. Finally, I would like to thank my family and my friends for their support over the years.

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω όλους εκείνους που συνέβαλαν στην ολοκλήρωση της εκπόνησης αυτής της εργασίας. Αρχικά θα ήθελα να ευχαριστήσω το επιβλέποντα καθηγητή μου Δημήτριο Κατσαρό που με ενέπνευσε και μου έδωσε την ευκαιρία να ασχοληθώ με μια πολύ ενδιαφέρουσα διπλωματική εργασία. Είμαι ευγνώμων στον Φιλήμων Γεωργίου, κύριο διαχειριστή συστημάτων στο Εθνικό Κέντρο Έρευνας και Τεχνολογικής Ανάπτυξης για την πολύτιμη γνώση και βοήθειά του κατά την υλοποήση της εργασίας . Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου και τους φίλους μου για την στήριξη τους όλα αυτά τα χρόνια.

Abstract

The purpose of this work is to compare the performance between Hadoop and Spark on some demanding and modern applications, such as iterative computation, real-time data processing and machine learning with Google's TensorFlow. The runtime architectures of both Spark and Hadoop will be compared to illustrate their differences, and the components of their ecosystems will be tabled to show their respective characteristics. In this case study, we will highlight the performance comparison between Spark and Hadoop as the growth of data size and iteration counts and show how to tune in Hadoop and Spark to achieve higher performance. In addition, there will be several appendixes which describes how to install and launch Hadoop and Spark, how to implement the three case studies using java and Scala programming, and how to verify the correctness of the running results.

1. Introduction

The world is growing at an enormous speed every moment, and at the same speed is growing the overall data size across the globe, which technically constitutes the term known as ‘Big Data’ The amount of newly generated data per year is huge and keeps on growing tremendously: from about 150 Exabytes in 2005 (worldwide) to approximately 1200 Exabytes in 2010. Nowadays, we create 2.5 quintillion bytes of data every day [0]. Twitter users generate over 500 million tweets every day, and a similar number of images is uploaded to Facebook. In 2016, the Facebook graph, which reflects the friendship relation between Facebook users, features more than a billion nodes and over hundreds of billions friendship edge5. And, the size of the indexed World Wide Web (estimated via the size of Googles index) is over 45 billion web pages6, and Google alone performs several billion searches on it every day.

A similar data explosion can be observed in the scientific world, for example in genetics, biology, or particle physics, as well as ever increasing digitized text collections. For instance, particles collide in the large hadron collider (LHC) detectors approximately 1 billion times per second, generating about one petabyte of collision data per second. Even though only the most “interesting” events can be stored and processed, CERN data center has already accumulated over 200 petabytes of filtered data.

Apart from big data, a yet another technological revolution that is taking the world by storm these days is ‘Artificial Intelligence’ or AI and ‘Machine Learning. The field was founded on the claim that human intelligence "can be so precisely described that a machine can be made to simulate it". In the twenty-first century, AI techniques have experienced a resurgence following concurrent advances in computer power, large amounts of data, and theoretical understanding; and AI techniques have become an essential part of the technology industry, helping to solve many challenging problems in computer science, software engineering and operations research.

1.1 TensorFlow

This chapter demonstrates a brief anatomy and analysis of TensorFlow library, Apache Hadoop and Apache Spark frameworks.
There will be a brief reference to their history and then we will analyze their functionality, their basic architecture and the substantial advantages of each technology separately

TensorFlow is a machine-learning library that uses data flow graphs to build models [1]. The main purpose of the library is to create models to solve various NLP and image recognition tasks. Starting in 2011, Google Brain built DistBelief as a proprietary machine learning system based on deep learning neural networks. Its use grew rapidly across diverse Alphabet companies in both research and commercial applications. Google assigned multiple computer scientists, to simplify and refactor the codebase of DistBelief into a faster, more robust application-grade library, which became TensorFlow. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays. These arrays are referred to as "tensors". TensorFlow was released under the Apache 2.0 open source license on November 9, 2015 and crossed over in both research and industry AI development

TensorFlow has been created for Deep Learning to let a user create a neural network architecture by himself (or herself, of course). Still, the library allows the user to work with statistical machine learning algorithms. However, it does not provide them out-of-the-box – user has to implement them on his own and TensorFlow provides only tools to do this.

As we can see in Figure 1, all the computations are represented as directed graphs, where the computations themselves (and input/output data as well) are nodes. The edges of the graph are paths, by which the data flows from node to node.

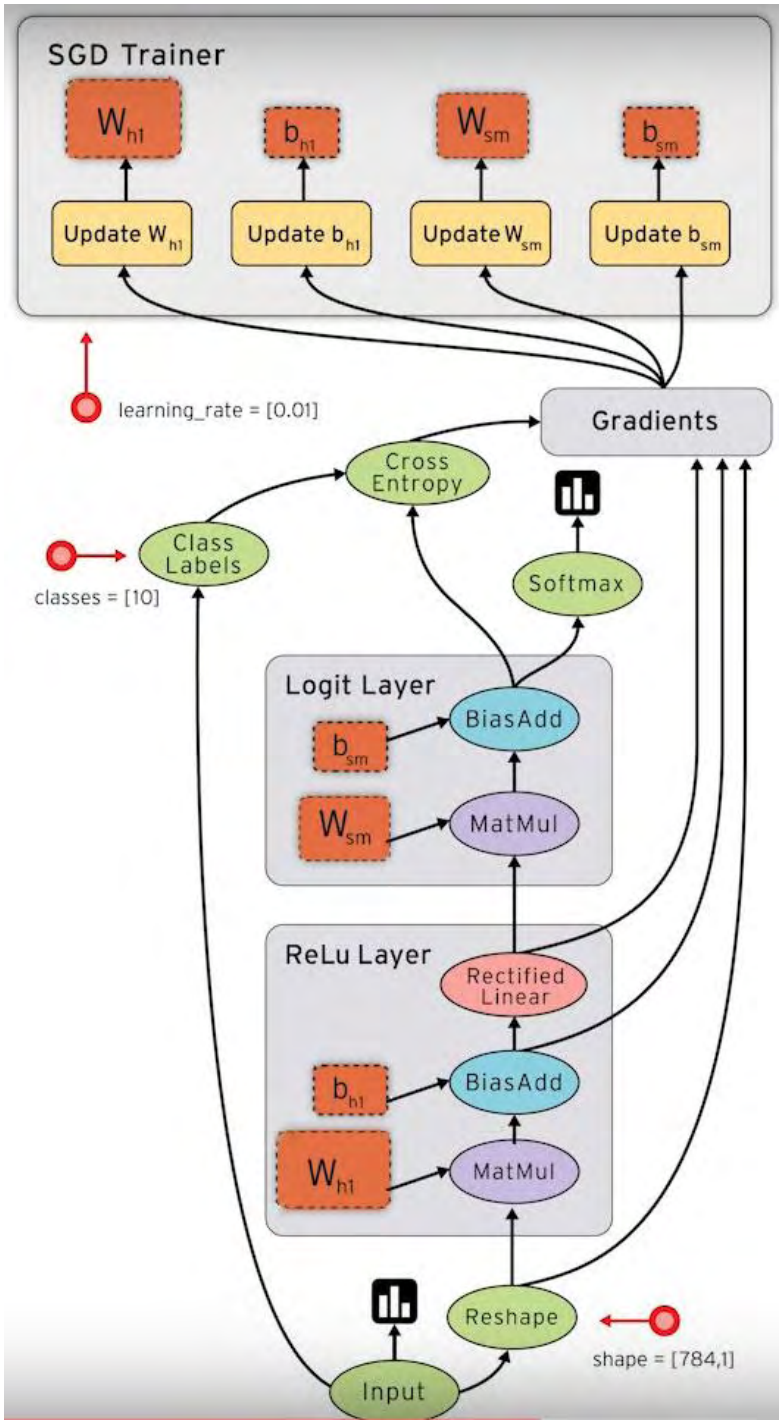


Figure 1:TensorFlow directed graph (source: adtmag.com/articles/2015/12/01/machine-learning-donation)

Data in TensorFlow are represented as tensors (multidimensional and dynamically sized data arrays) [2]. Tensors flow in the graph from node to node, thus making the name of the library sound logical. Simply speaking, a tensor is a 3D matrix (but it is not a strict mathematical definition, of course!). On a figure below, you may see a tensor in terms of vivisection. Comparing to

matrix it has more degrees of freedom regarding data selection and slicing. The following figure (Figure 2) demonstrates TensorFlow's Fibers and Slices:

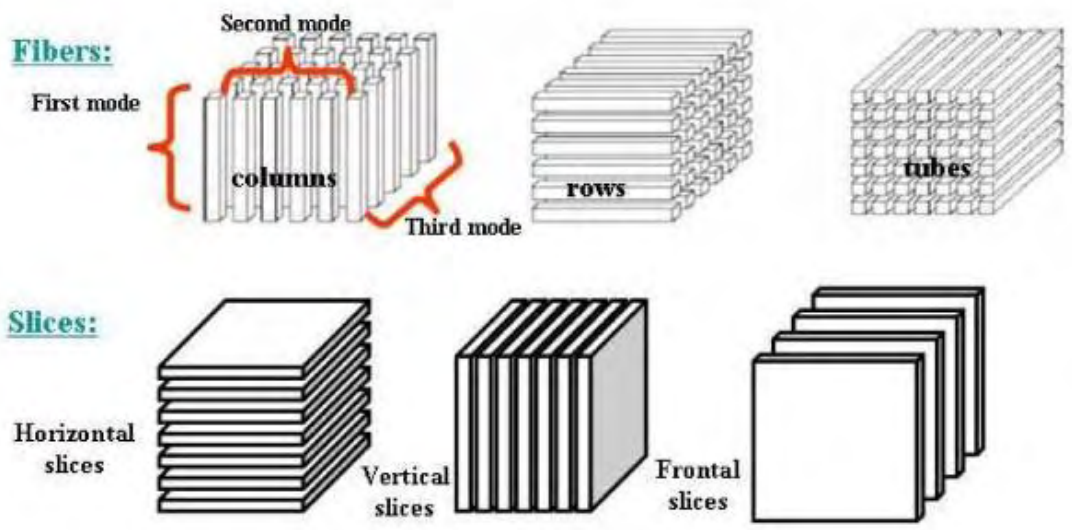


Figure 2:TensorFlow Fibers and Slices (source: www.researchgate.net)

Some of the main features and advantages of TensorFlow are:

- **Portability:** TensorFlow has made it possible to play around an idea on your laptop without having any other hardware support. It runs on GPUs, CPUs, desktops, servers, and mobile computing platforms. You can deploy a trained model on your mobile as a part of your product, and that's how it serves as a true portability feature.

- **Flexibility:** You need to express your computation as a data flow graph to use TensorFlow. It is a highly flexible system which provides multiple models or multiple versions of the same model can be served simultaneously. The architecture of TensorFlow is highly modular, which means you can use some parts individually or can use all the parts together. Such flexibility facilitates non-automatic migration to new models/versions, A/B testing experimental models, and canarying new models.

- **Performance:** TensorFlow allows you to make the most of your available hardware with its advanced support for threads, asynchronous computation, and queues. Just assign compute elements of your TensorFlow graph to different devices and let it manage the copies itself. It also facilitates you with the language options to execute your computational graph. TensorFlow iPython notebook helps in keeping codes, notes, and visualization in a logically grouped and interactive style.

- **Research and Production:** It can be used to train and serve models in live mode to real customers. To put it simply, rewriting codes is not required and the industrial researchers can apply their ideas to products faster. Also, academic researchers can share codes directly with greater reproducibility. In this way it helps to carry out research and production processes faster.

- **Auto Differentiation:** It has automatic differentiation capabilities which benefits gradient based machine learning algorithms. You can define the computational architecture of your predictive model, combine it with your objective function and add data to it- TensorFlow manages derivatives computing processes automatically. You can compute the derivatives of some values with respect to some other values results in graph extension and you can see exactly what’s happening.

1.2 Hadoop

Apache Hadoop is a collection of open-source software utilities that facilitate using a network of many computers to solve problems involving massive amounts of data and computation [3]. It provides a software framework for distributed storage and processing of big data using the MapReduce programming model, designed to run on commodity hardware, it has also found use on clusters of higher-end hardware. According to its co-founders, Doug Cutting and Mike Cafarella, the genesis of Hadoop was the "Google File System" paper that was published in October 2003.[4] This paper spawned another one from Google – "MapReduce: Simplified Data Processing on Large Clusters".[5] Development started on the Apache Nutch project, but was moved to the new Hadoop subproject. Doug Cutting, who was working at Yahoo at the time, named it after his son's toy elephant.

The core of Apache Hadoop consists of a storage part, known as Hadoop Distributed File System (HDFS), and a processing part which is a MapReduce programming model. Hadoop splits files into large blocks and distributes them across nodes in a cluster. It then transfers packaged code into nodes to process the data in parallel. This approach takes advantage of data locality, where nodes manipulate the data they have access to. This allows the dataset to be processed faster and more efficiently than it would be in a more conventional supercomputer architecture that relies on a parallel file system where computation and data are distributed via high-speed networking. HDFS has many similarities with existing distributed file systems. However, the differences from other distributed file systems are significant. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. HDFS relaxes a few POSIX requirements to enable streaming access to file system data.

The base Apache Hadoop framework is composed of the following modules:

- Hadoop Common – contains libraries and utilities needed by other Hadoop modules;
- Hadoop Distributed File System (HDFS) – a distributed file-system that stores data on commodity machines, providing very high aggregate bandwidth across the cluster;
- Hadoop YARN – introduced in 2012 is a platform responsible for managing computing resources in clusters and using them for scheduling users' applications
- Hadoop MapReduce – an implementation of the MapReduce programming model for large-scale data processing.

Hadoop consists of the Hadoop Common package, which provides file system and operating system level abstractions, a MapReduce engine (either MapReduce or YARN) and the Hadoop Distributed File System (HDFS). The Hadoop Common package contains the Java Archive (JAR) files and scripts needed to start Hadoop.

For effective scheduling of work, every Hadoop-compatible file system should provide location awareness – the name of the rack (or, more precisely, of the network switch) where a worker node is. Hadoop applications can use this information to execute code on the node where the data is, and, failing that, on the same rack/switch to reduce backbone traffic. HDFS uses this method when replicating data for data redundancy across multiple racks. This approach reduces the impact of a rack power outage or switch failure; if any of these hardware failures occurs, the data will remain available.

A small Hadoop cluster includes a single master and multiple worker nodes. The master node consists of a Job Tracker, Task Tracker, NameNode, and DataNode. A slave or worker node acts as both a DataNode and TaskTracker, though it is possible to have data-only and compute-only worker nodes. These are normally used only in nonstandard applications. In a larger cluster, HDFS nodes are managed through a dedicated NameNode server to host the file system index, and a secondary NameNode that can generate snapshots of the namenode's memory structures, thereby preventing file-system corruption and loss of data. Similarly, a standalone JobTracker server can manage job scheduling across nodes. When Hadoop MapReduce is used with an alternate file system, the NameNode, secondary NameNode, and DataNode architecture of HDFS are replaced by the file-system-specific equivalents. The following figures (Figure 3, Figure 4 and Figure 5) shows HDFS architecture, a MapReduce sequence diagram and a Hadoop ecosystem respectively:

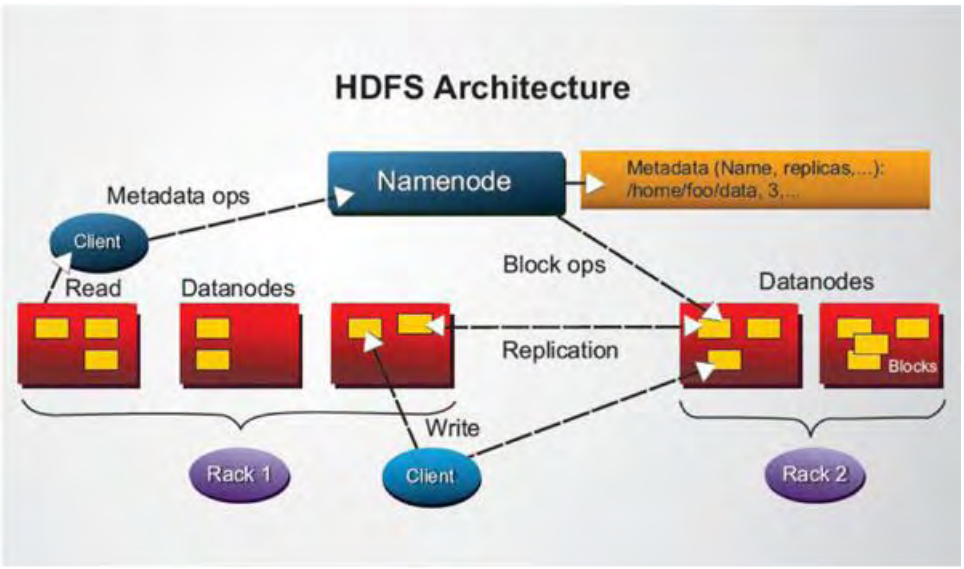


Figure 3:HDFS Architecture source: http://spmarchitecture.com)

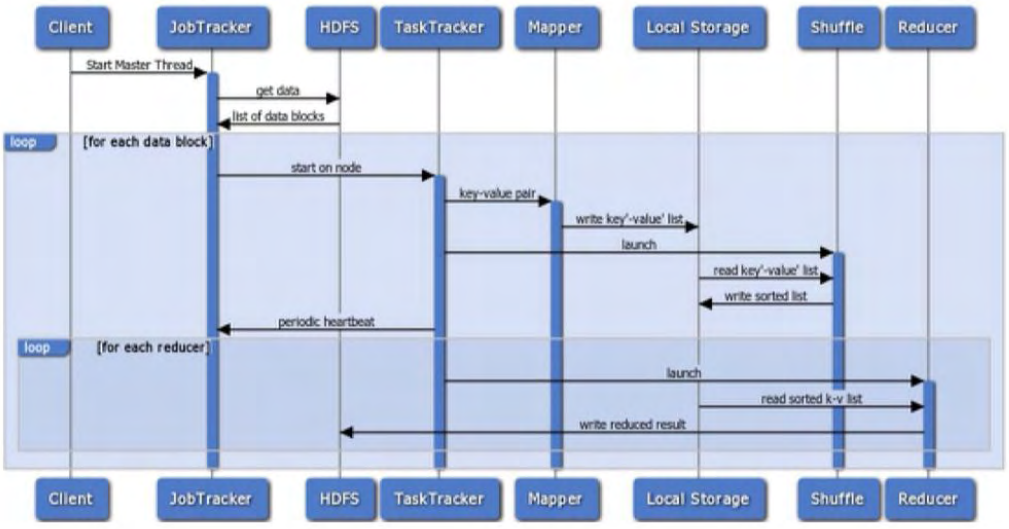


Figure 4:Map Reduce Sequence Diagram source: http://broadwaycomputers.us)

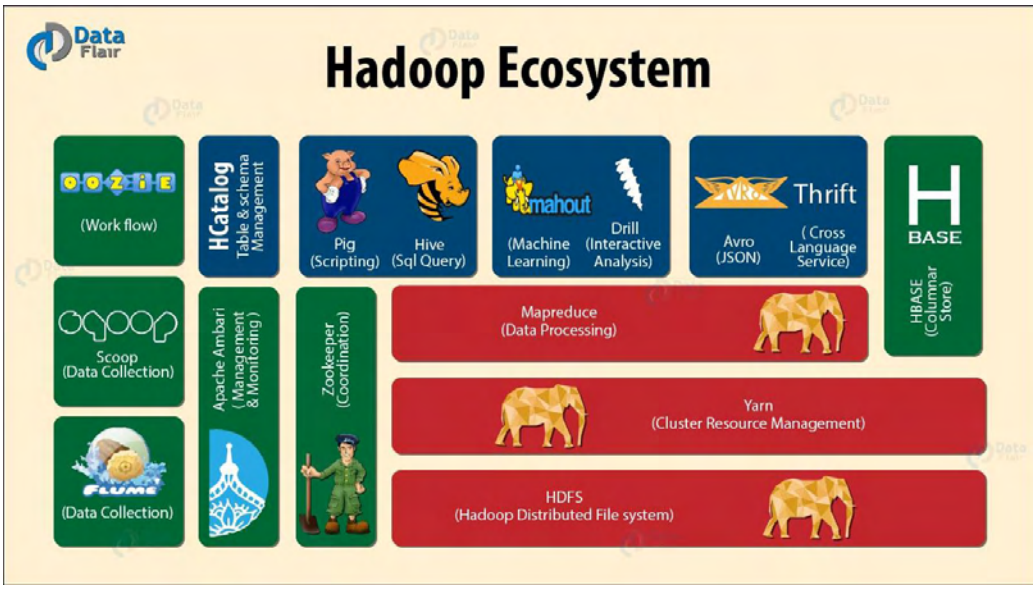


Figure 5:Hadoop ecosystem example (source: wr.informatik.uni-hamburg.de)

Some of the main features and advantages of Apache Hadoop are [6]:

- **Scalable**

Hadoop is a highly scalable storage platform, because it can store and distribute very large data sets across hundreds of inexpensive servers that operate in parallel. Unlike traditional relational database systems (RDBMS) that can't scale to process large amounts of data, Hadoop enables businesses to run applications on thousands of nodes involving thousands of terabytes of data.

- **Cost effective**

Hadoop also offers a cost effective storage solution for businesses' exploding data sets. The problem with traditional relational database management systems is that it is extremely cost prohibitive to scale to such a degree in order to process such massive volumes of data. In an effort to reduce costs, many companies in the past would have had to down-sample data and classify it based on certain assumptions as to which data was the most valuable. The raw data would be deleted, as it would be too cost-prohibitive to keep. While this approach may have worked in the short term,

this meant that when business priorities changed, the complete raw data set was not available, as it was too expensive to store. The cost savings are staggering: instead of costing thousands to tens of thousands of pounds per terabyte, Hadoop offers computing and storage capabilities for hundreds of pounds per terabyte.

- **Flexible**

Hadoop enables businesses to easily access new data sources and tap into different types of data (both structured and unstructured) to generate value from that data. This means businesses can use Hadoop to derive valuable business insights from data sources such as social media, email conversations or clickstream data. In addition, Hadoop can be used for a wide variety of purposes, such as log processing, recommendation systems, data warehousing, market campaign analysis and fraud detection.

- **Fast**

Hadoop's unique storage method is based on a distributed file system that basically 'maps' data wherever it is located on a cluster. The tools for data processing are often on the same servers where the data is located, resulting in much faster data processing. If you're dealing with large volumes of unstructured data, Hadoop is able to efficiently process terabytes of data in just minutes, and petabytes in hours.

- **Resilient to failure**

A key advantage of using Hadoop is its fault tolerance. When data is sent to an individual node, that data is also replicated to other nodes in the cluster, which means that in the event of failure, there is another copy available for use.

The MapReduce distribution goes beyond that by eliminating the NameNode and replacing it with a distributed No NameNode architecture that provides true high availability. Our architecture provides protection from both single and multiple failures.

1.3 Spark

Spark is a cluster computing framework and an engine for large-scale data processing. It constructs a distributed collections of objects, resilient distributed datasets (RDDs) in memory, and then performs a variety of operations in parallel on these datasets. Spark greatly outperforms Hadoop MapReduce by 10x in iterative machine learning tasks [7] and is up to20x faster for iterative applications [8] Apache Spark is considered as a powerful complement to Hadoop. Big data's original technology of choice Spark is a general-purpose data processing engine that is suitable for use in a wide range of circumstances. Application developers and data scientists incorporate Spark into their applications to rapidly query, analyze, and transform data at scale. Tasks most frequently associated with Spark include interactive queries across large data sets, processing of streaming data from sensors or financial systems, and machine learning tasks.

Spark began life in 2009 as a project within the AMPLab at the University of California, Berkeley. More specifically, it was born out of the necessity to prove out the concept of Mesos, which was also created in the AMPLab. Spark was first discussed in the Mesos white paper *Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center*, written most notably by Benjamin Hindman and Matei Zaharia [5].

Spark became an incubated project of the Apache Software Foundation in 2013, and it was promoted early in 2014 to become one of the Foundation's top-level projects [9]. Spark is currently one of the most active projects managed by the Foundation, and the community that has grown up around the project includes both prolific individual contributors and well-funded corporate backers such as Databricks, IBM, and China's Huawei.

Spark can handle several petabytes of data at a time, distributed across a cluster of thousands of cooperating physical or virtual servers. It has an extensive set of developer libraries and APIs and supports languages such as Java, Python, R, and Scala; its flexibility makes it well-suited for a range of use cases. Spark is often used alongside Hadoop's data storage module—HDFS—but it can integrate equally well with other popular data storage

subsystems such as HBase, Cassandra, Map-DB, MongoDB and Amazon's S3.

Apache Spark has a well-defined and layered architecture where all the spark components and layers are loosely coupled and integrated with various extensions and libraries. Apache Spark Architecture is based on two main abstractions: Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG)

Resilient Distributed Datasets

RDD's are collection of data items that are split into partitions and can be stored in-memory on workers nodes of the spark cluster. In terms of datasets, apache spark supports two types of RDD's – Hadoop Datasets which are created from the files stored on HDFS and parallelized collections which are based on existing Scala collections. Spark RDD's support two different types of operations – Transformations and Actions [10].

Directed Acyclic Graph

Direct - Transformation is an action which transitions data partition state from A to B. Acyclic -Transformation cannot return to the older partition

DAG is a sequence of computations performed on data where each node is an RDD partition and edge is a transformation on top of data. The DAG abstraction helps eliminate the Hadoop MapReduce multi0stage execution model and provides performance enhancements over Hadoop.

Apache Spark (as you can see in Figure 6)follows a master/slave architecture with two main daemons and a cluster manager:

i. Master Daemon – (Master/Driver Process)

ii. Worker Daemon –(Slave Process)

A spark cluster has a single Master and any number of Slaves/Workers. The driver and the executors run their individual Java processes and users can run them on the same horizontal spark cluster or on separate machines i.e. in a vertical spark cluster or in mixed machine configuration(Figure 7).

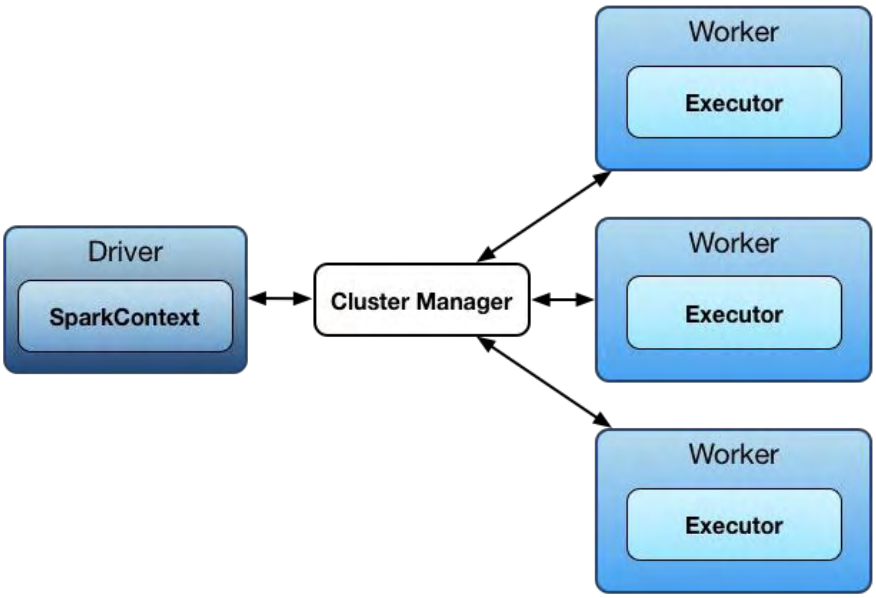


Figure 6:Spark Architecture (source: siliconangle.com)

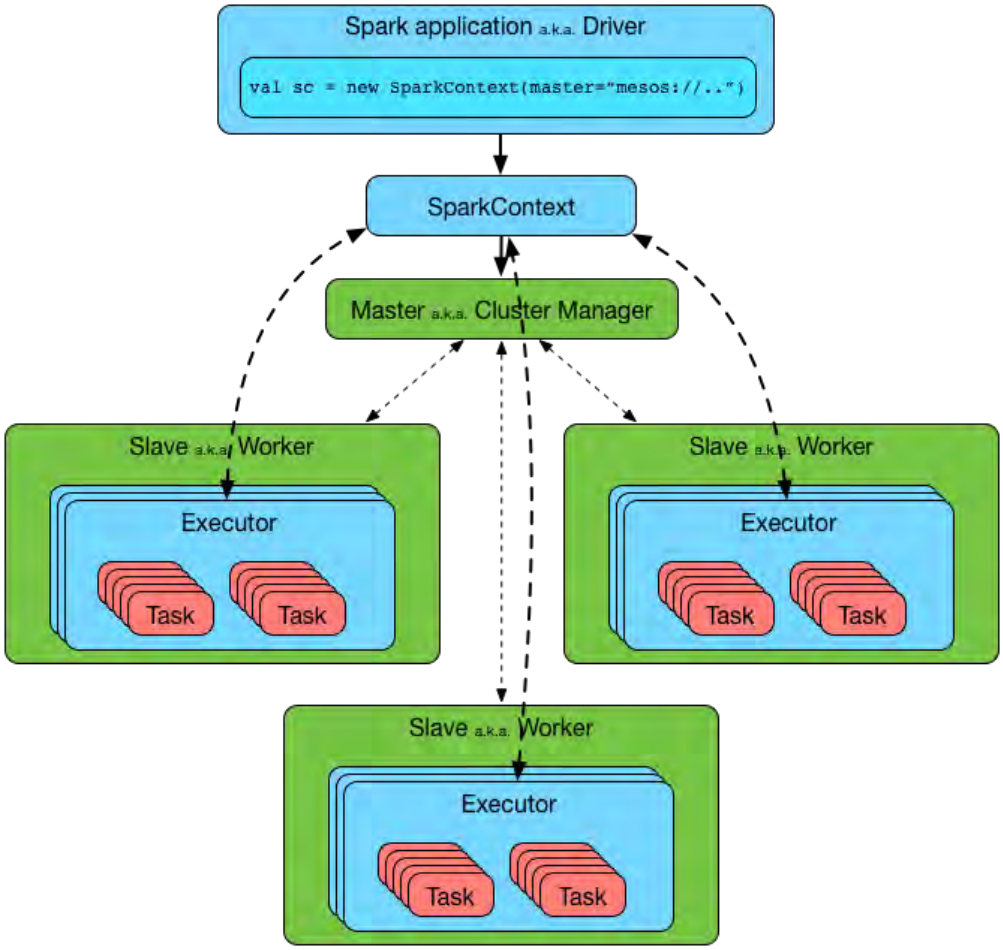


Figure 7:Spark Architecture (clusters) (source: databricks.com)

Role of Spark driver:

Spark Driver – Master Node of a Spark Application is the central point and the entry point of the Spark Shell (Scala, Python, and R). The driver program runs the main () function of the application and is the place where the Spark Context is created. Spark Driver contains various components – DAGScheduler, TaskScheduler, BackendScheduler and BlockManager responsible for the translation of spark user code into actual spark jobs executed on the cluster.

- The driver program that runs on the master node of the spark cluster schedules the job execution and negotiates with the cluster manager.
- It translates the RDD's into the execution graph and splits the graph into multiple stages.
- Driver stores the metadata about all the Resilient Distributed Databases and their partitions.
- Cockpits of Jobs and Tasks Execution -Driver program converts a user application into smaller execution units known as tasks. Tasks are then executed by the executors i.e. the worker processes which run individual tasks.
- Driver exposes the information about the running spark application through a Web UI at port 4040

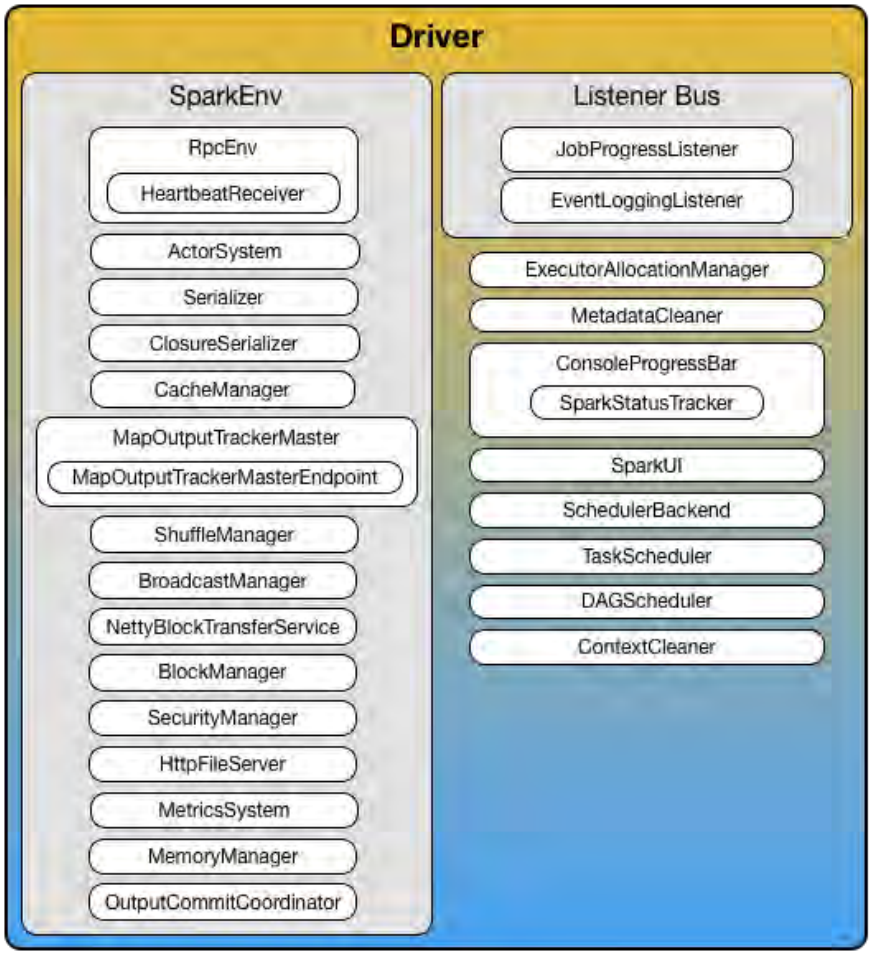


Figure 8:Spark Driver with services(source:stackoverflow.com)

Role of Executor in Spark:

Executor is a distributed agent responsible for the execution of tasks. Every spark applications have its own executor process. Executors usually run for the entire lifetime of a Spark application and this phenomenon is known as “Static Allocation of Executors”. However, users can also opt for dynamic allocations of executors wherein they can add or remove spark executors dynamically to match with the overall workload.

As shown in Figure 9:

- Executor performs all the data processing.
- Reads from and Writes data to external sources.
- Executor stores the computation results data in-memory, cache or on hard disk drives.
- Interacts with the storage systems.

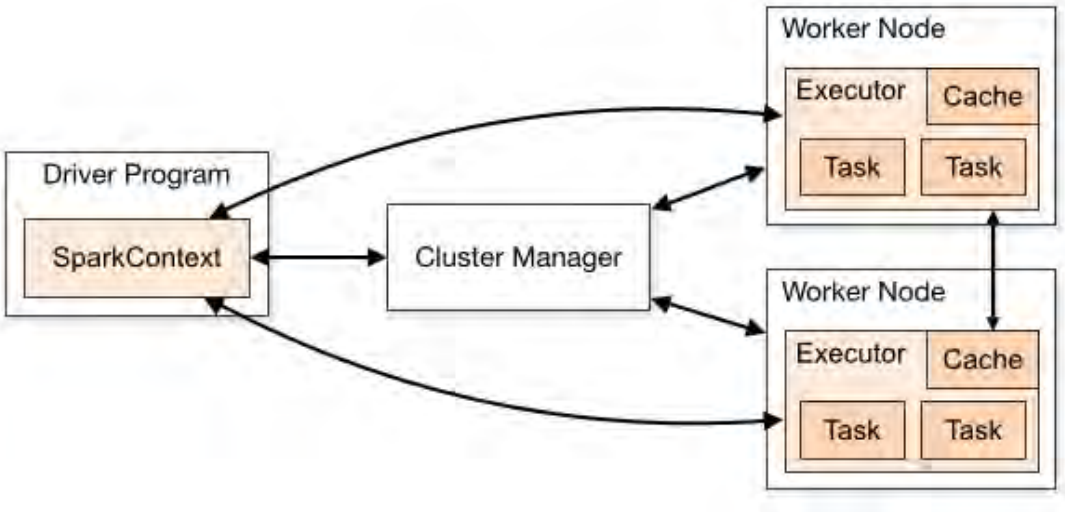


Figure 9:Spark architecture(executors)(source:databricks.com)

Role of Cluster Manager in Spark Architecture

Cluster Manager (as shown in Figure 10)is an external service responsible for acquiring resources on the spark cluster and allocating them to a spark job. There are 3 different types of cluster managers a Spark application can leverage for the allocation and deallocation of various physical resources such as memory for client spark jobs, CPU memory, etc. Hadoop YARN, Apache Mesos or the simple standalone spark cluster manager either of them can be launched on-premise or in the cloud for a spark application to run.

Choosing a cluster manager for any spark application depends on the goals of the application because all cluster managers provide different set of scheduling capabilities. Specifically, to run on a cluster, the SparkContext can connect to several types of cluster managers (either Spark's own standalone cluster manager, Mesos or YARN), which allocate resources across applications. Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for your application. Next, it sends your application code (defined by JAR or Python files passed to SparkContext) to the executors. Finally, SparkContext sends tasks to the executors to run.

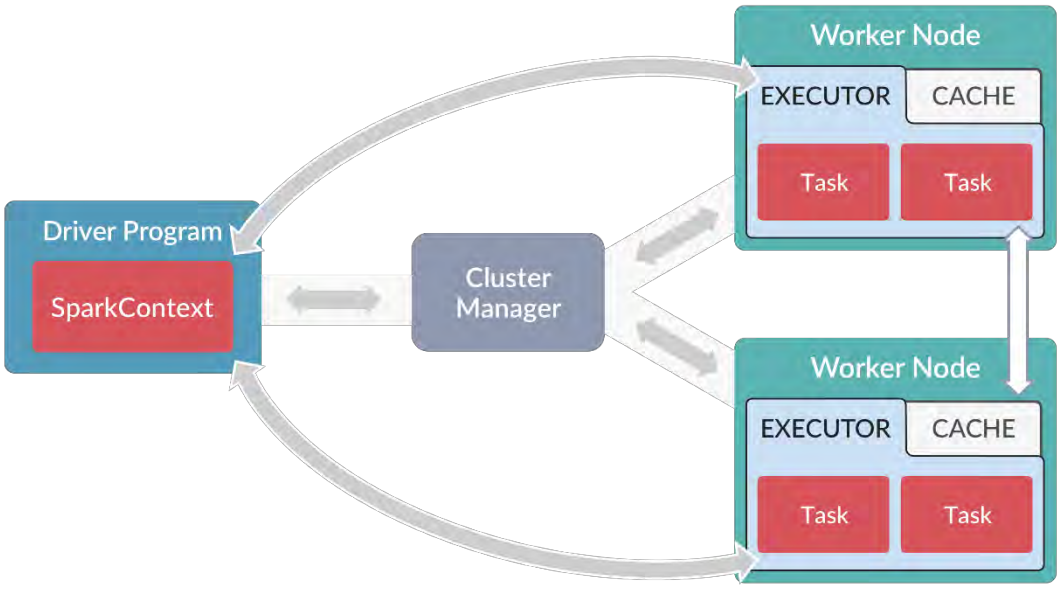


Figure 10:Spark Architecture (Cluster managers)(source:dzone.com)

Some of the key features and advantages Apache Hadoop are:

- **Simplicity**

Spark's capabilities are accessible via a set of rich APIs, all designed specifically for interacting quickly and easily with data at scale. These APIs are well documented and structured in a way that makes it straightforward for data scientists and application developers to quickly put Spark to work.

- **Speed**

Spark is designed for speed, operating both in memory and on disk. Using Spark, a team of people from Databricks tied for first place with a team from University of California, San Diego in the 2014 Daytona Gray Sort 100TB Benchmark challenge. The challenge involves processing a static data set;

the Databricks team was able to process 100 terabytes of data stored on solid-state drives in just 23 minutes, and the previous winner took 72 minutes by using Hadoop and a different cluster configuration. Spark can perform even better when supporting interactive queries of data stored in memory. In those situations, there are claims that Spark can be 100 times faster than Hadoop's MapReduce.

- **Support**

Spark supports a range of programming languages, including Java, Python, R, and Scala. Although often closely associated with HDFS, Spark includes native support for tight integration with a number of leading storage solutions in the Hadoop ecosystem and beyond. Furthermore, the Apache Spark community is large, active, and international. A growing set of commercial providers including Databricks, IBM, and all of the main Hadoop vendors deliver comprehensive support for Spark-based solutions.

2. Hadoop Vs Spark: A head to head comparison

2.0 Setup and Prerequisites of an HPC Cluster

The experimental cluster used consists of five computer systems. One of them serves as a manager (both a master and a slave node but with no acceleration capabilities). The other four are designed to be slave nodes. The hardware information for the cluster is shown as following:

- 4 nodes interconnected by Each node 10G-Ethernet.
- Each node has 2 Intel Xeon CPU E5-2620v4 running at 2.10GHz
- Each CPU has 8 cores, each core has 2 threads (hyper-threading).
- Each node has 128GB of memory.
- The configured capacity for HDFS is 6 TB with 1.4 TB per node.

We use the CentOS 7.4 operating system and JAVA 1.8.0 version for all the nodes. We use Hadoop 2.8.2 (stable) Spark 2.2 and YARN 1.3.0 for the resource management layer of Hadoop. The version for Hadoop and Spark are stable released and the version for Yarn is the latest release.

The following guide demonstrates all the necessary commands to install Spark and Hadoop for usage with multiple clusters in a high-performance system.

For Hadoop, we use Apache Hadoop YARN. Apache Yarn – “Yet Another Resource Negotiator” is the resource management layer of Hadoop [11] . The Yarn was introduced in Hadoop 2.x. versions Yarn allows different data processing engines like graph processing, interactive processing, stream processing as well as batch processing to run and process data stored in HDFS (Hadoop Distributed File System). Apart from resource management, Yarn is also used for job Scheduling. Yarn extends the power of Hadoop to

other evolving technologies, so they can take the advantages of HDFS (most reliable and popular storage system on the planet) and economic cluster.

Apache Yarn is also considered as the data operating system for Hadoop 2. x. versions. The yarn-based architecture of Hadoop 2.x provides a general-purpose data processing platform which is not just limited to the MapReduce. It enables Hadoop to process other purpose-built data processing system other than MapReduce. It allows running several different frameworks on the same hardware where Hadoop is deployed.

To install Yarn use Debian package repositories but firstly you need to configure the repository:

```
curl -sS https://dl.yarnpkg.com/debian/pubkey.gpg | sudo apt-key add -

echo "deb https://dl.yarnpkg.com/debian/ stable main" | sudo tee /etc/apt/sources.list.d/yarn.list
```

Then you can simply:

```
sudo apt-get update && sudo apt-get install yarn
```

Test that Yarn is installed by running:

```
yarn --version
```

For Hadoop 2.8.2 version:

Before you proceed to the installation, create a normal user for the install, and a user called hadoop for any Hadoop daemons. Do **not** create SSH keys for hadoop users. SSH keys will be addressed in a later section.

First, make your network adapter as Bridge or Host-Only depending upon your requirements. You can use virtual machines (using each machine's respectively) but in our case (physical machine cluster) we have created a cluster with 4 nodes. For each node to communicate with its names, edit

the /etc/hosts file to add the IP address of the servers (in case you have multiple machines)

The master node will use an ssh-connection to connect to other nodes with key-pair authentication, to manage the cluster. Login to node-master as the hadoop user, and generate an ssh-key:

```
ssh-keygen -b 4096
```

Copy the key to the other nodes. It's good practice to also copy the key to the **node-master** itself, so that you can also use it as a DataNode if needed. Type the following commands, and enter the hadoop user's password when asked. If you are prompted whether or not to add the key to known hosts, enter yes:

```
ssh-copy-id -i $HOME/.ssh/id_rsa.pub hadoop@node-master

ssh-copy-id -i $HOME/.ssh/id_rsa.pub hadoop@node1

ssh-copy-id -i $HOME/.ssh/id_rsa.pub hadoop@node2

ssh-copy-id -i $HOME/.ssh/id_rsa.pub hadoop@node3

ssh-copy-id -i $HOME/.ssh/id_rsa.pub hadoop@node4
```

Login to **node-master** as the hadoop user, download the Hadoop tarball from Hadoop project page, and unzip it:

```
cd <desired folder>

wget http://apache.mindstudios.com/hadoop/common/hadoop-2.8.1/hadoop-2.8.1.tar.gz
```

```
tar -xzf hadoop-2.8.1.tar.gz

mv hadoop-2.8.1 hadoop
```

Add Hadoop binaries to your PATH. Edit `/home/hadoop/.profile` and add the following line:

```
PATH=/home/hadoop/hadoop/bin:/home/hadoop/hadoop/sbin:$PATH
```

Configure the Master Node. Configuration will be done on **node-master** and replicated to other nodes:

Get your Java installation path. If you installed open-jdk from your package manager, you can get the path with the command:

```
update-alternatives --display java
```

Take the value of the current link and remove the trailing /bin/java. For example, on Debian, the link is `/usr/lib/jvm/java-8-openjdk-amd64/jre/bin/java`, so `JAVA_HOME` should be `/usr/lib/jvm/java-8-openjdk-amd64/jre`.

If you installed java from Oracle, JAVA_HOME is the path where you unzipped the java archive.

Edit `~/hadoop/etc/hadoop/hadoop-env.sh` and replace this line:

```
export JAVA_HOME=${JAVA_HOME}
```

with your actual java installation path. For example, on a Debian with open-jdk-8:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre
```

Set NameNode Location:

On each node update `~/hadoop/etc/hadoop/core-site.xml` you want to set the NameNode location to **node-master** on port `9000`:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
  <configuration>
    <property>
      <name>fs.default.name</name>
      <value>hdfs://node-master:9000</value>
    </property>
  </configuration>
```

Set path for HDFS: Edit `hdfs-site.conf`:

```
<configuration>
  <property>
      <name>dfs.namenode.name.dir</name>
      <value>/home/hadoop/data/nameNode</value>
  </property>
```

```
    <property>
        <name>dfs.datanode.data.dir</name>
        <value>/home/hadoop/data/dataNode</value>
    </property>

    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

The last property, `dfs.replication`, indicates how many times data is replicated in the cluster. You can set `4` to have all the data duplicated on the four nodes. Don't enter a value higher than the actual number of slave nodes.

Set Yarn as Job Scheduler:

In `~/hadoop/etc/hadoop/`, rename `mapred-site.xml.template` to `mapred-site.xml`:

```
cd ~/hadoop/etc/hadoop

mv mapred-site.xml.template mapred-site.xml
```

Edit the file, setting yarn as the default framework for MapReduce operations:

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

Configure Yarn:

Edit `yarn-site.xml`:

```
<configuration>
    <property>
        <name>yarn.acl.enable</name>
        <value>0</value>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname</name>
        <value>node-master</value>
    </property>
```

```
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
</configuration>
```

Configure Slaves

The file `slaves` is used by startup scripts to start required daemons on all nodes. Edit `~/hadoop/etc/hadoop/slaves` to be:

```
node1
node2
node3
node4
```

For Spark 2.2 version:

Install Spark on Master:

Add entries in host files

```
sudo nano /etc/hosts
```

Add entries of master and slaves:

```
MASTER -<Corresponding IP>
SLAVE01-<Corresponding IP>
```

```
SLAVE01-<Corresponding IP>
```

Install Scala

```
sudo apt-get install scala
```

Configure SSH

Install Open SSH Server-Client

```
sudo apt-get install openssh-server openssh-client
```

Generate Key pairs

```
ssh-keygen -t rsa -P ""
```

Configure passwordless SSH

Copy the content of .ssh/id_rsa.pub (of master) to .ssh/authorized_keys (of all the slaves as well as master)

Check by SSH to all the Slaves

```
ssh slave01

ssh slave02

ssh slave03

ssh slave04
```

Download Spark from the official site and unrar tarball.

Setup Configuration:

Edit .bashrc

Edit .bashrc file located in user's home directory and add following environment variables:

```
export JAVA_HOME=<path-of-Java-installation> (eg: /usr/lib/jvm/java-8-oracle/)

export SPARK_HOME=<path-to-the-root-of-your-spark-installation> (eg: /home/dataflair/spark-2.2.0-bin-hadoop2.8.2/)

export PATH=$PATH:$SPARK_HOME/bin
```

Edit spark-env.sh:

Now edit configuration file spark-env.sh (in $SPARK_HOME/conf/) and set following parameters:

Note: Create a copy of template of spark-env.sh and rename it:

```
cp spark-env.sh.template spark-env.sh

export JAVA_HOME=<path-of-Java-installation> (eg: /usr/lib/jvm/java-8-oracle/)

export SPARK_WORKER_CORES=8
```

Add Slaves

Create configuration file slaves (in $SPARK_HOME/conf/) and add following entries:

```
slave01

slave02
```

Setup Prequisites on all the slaves

Run following steps on all the slaves (or worker nodes):

1. Add Entries in hosts file

2. Install Java 8

3. Install Scala

Start Spark Cluster:

Start Spark Services

```
sbin/start-all.sh
```

Check whether services have been started

Check daemons on Master

```
jps

Worker
```

Check daemons on Slaves

```
jps

Worker
```

The following Pictures are indicative instances of our experimental machine’s setup:

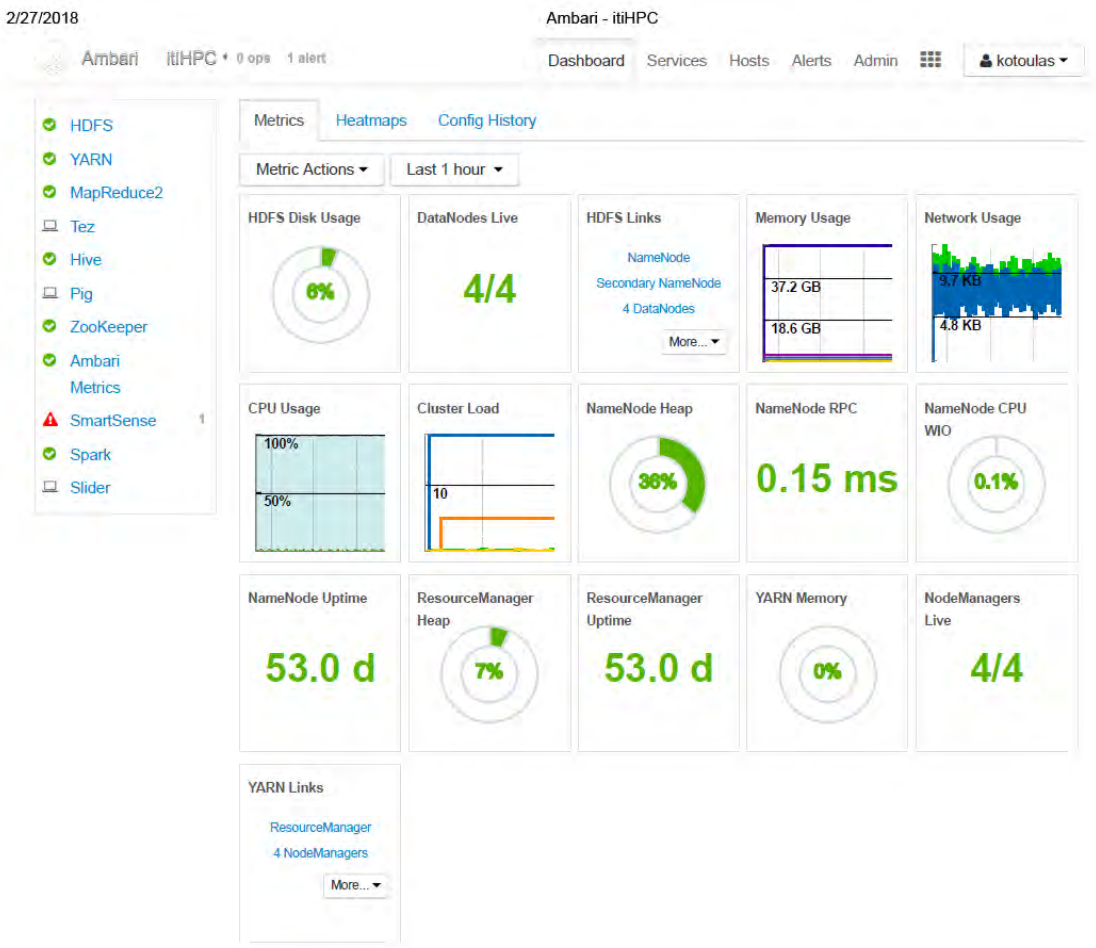


Figure 11:Cluster's Dashboard

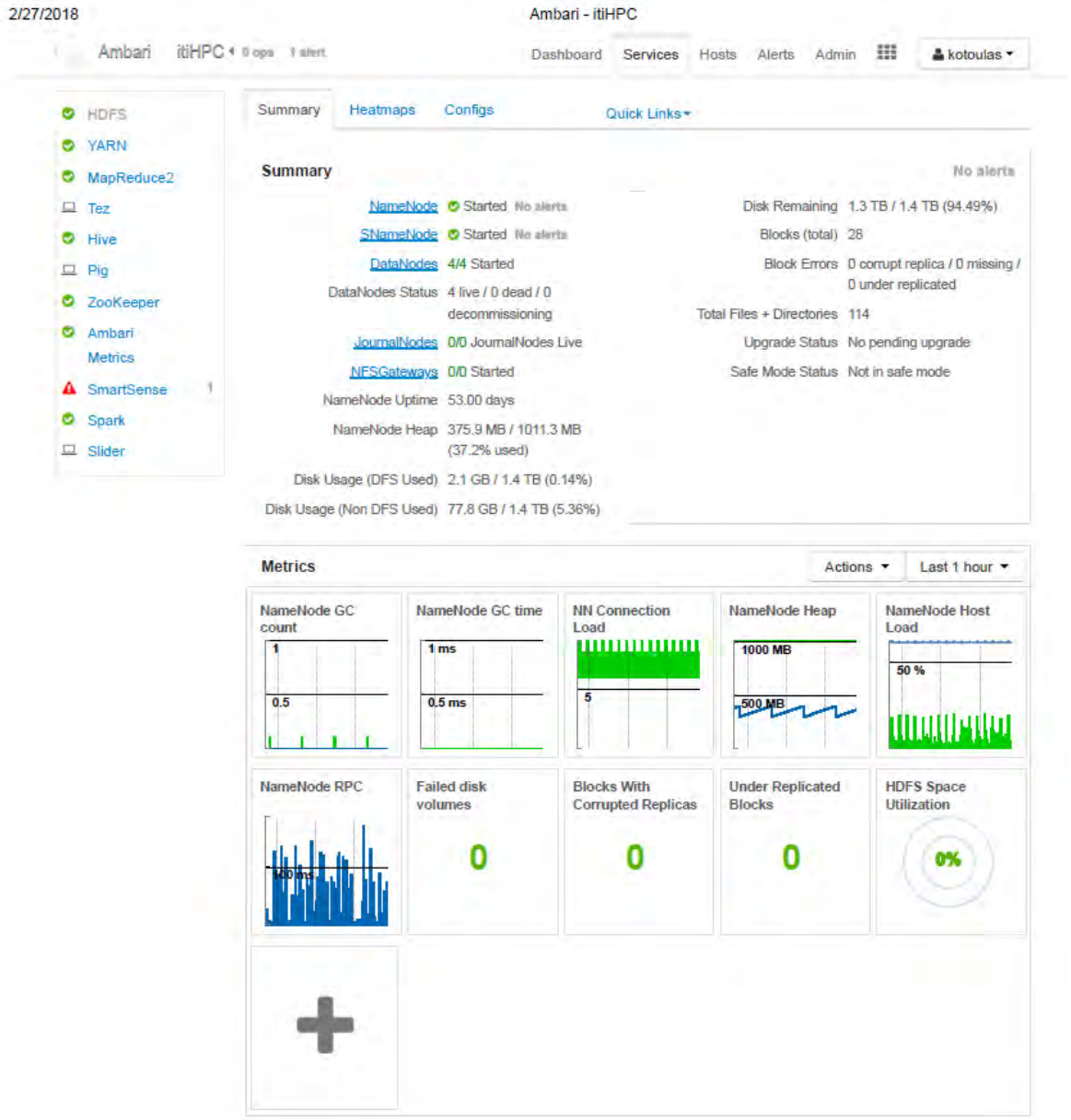


Figure 12:Cluster's Summary and Metrics for each Node

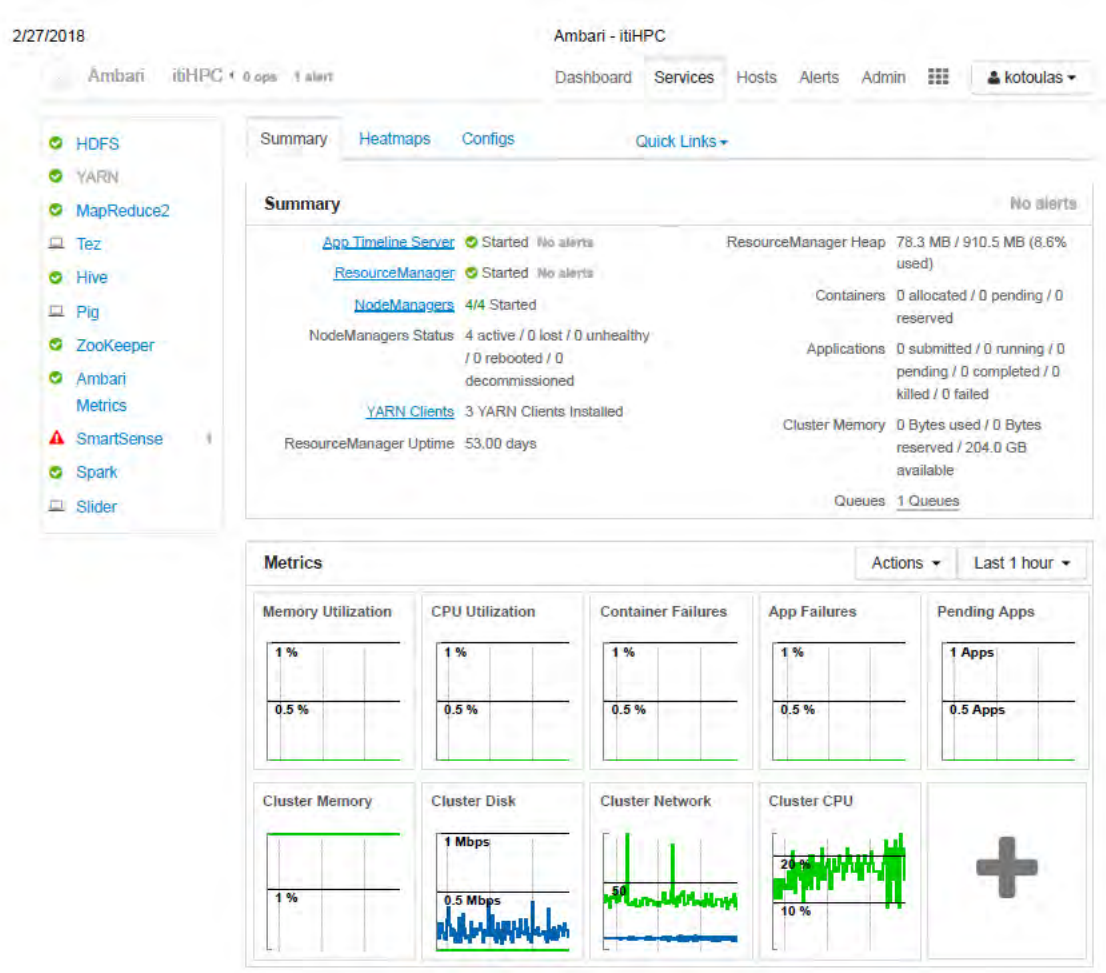


Figure 13:Manger node's Services and Metrics monitor

2.1 Evaluation and Results

We set up Hadoop and Spark clusters to run three demanding benchmarks including TeraSort, Naive Bayes and K-means[12][13][14][15]. These benchmarks are classified into two categories that are micro benchmark and machine learning. Both distributed systems are built on a five (1 manager-master/slave and 4 slaves) nodes cluster. We measured the running time, speedup, throughput, maximum and average memory and CPU usage for all the benchmarks on both platforms of Hadoop and Spark. Finally, we compared the performance differences among these two platforms based on the characteristics of the benchmarks. Also, the experimental results are shown in following chapters and analyzed separately for different benchmarks. We compared the performances between Hadoop and Spark on TeraSort, Naive Bayes and K-means. The experimental results showed that Spark is faster than Hadoop. Specifically, Spark has a outstanding performance on machine learning applications including K-means and Naïve Bayes since these applications apply a function repeatedly to the same dataset. For TeraSort, Spark runs faster with large input. However, Spark consumes more memory capacity and the performance for Spark is restricted by the memory.

2.1.1 TeraSort

Before proceeding to a ML benchmark, we wanted to measure and compare the sorting times between the two platforms. To achieve this, we selected to test TeraSort benchmark [15]. TeraSort is a popular benchmark that measures the amount of time to sort one terabyte of randomly distributed data on a given computer system. It is commonly used to measure MapReduce performance of an Apache™ Hadoop® cluster and in addition a reliable and dependable TeraSort version for Spark is disposable. It is written by Owen O'Malley at Yahoo Inc. and won the annual general-purpose terabyte sort benchmark in 2008 and 2009. The TeraSort package includes three applications: Teragen which is a MapReduce program and can be used to generate input data, TeraSort which can be used to sorts the input data, and TeraValidata which can be used to check the output.

The following Graph represents a running time performance comparison for running Sort between Apache Hadoop and Apache Spark.

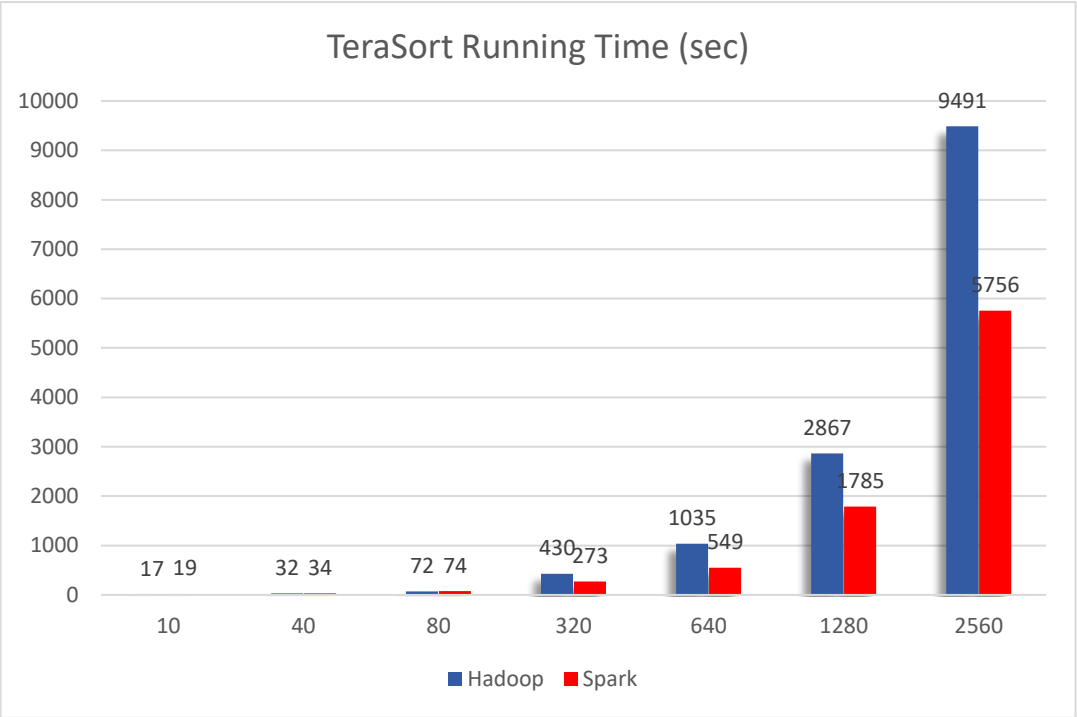


Figure 14: TeraSort Running Time Comparison for Hadoop and Spark

The input sizes range from 10 million to 2560 million records and the sizes range from 1 Gb to 2560 Gb with 100 bytes for each record.

Input	10M	40Gb	80Gb	320Gb	640Gb	1280Gb	2560Gb
Hadoop	17	32	73	430	1035	2867	9491
Spark	19	34	74	273	549	1785	5756
Speedup	**0.90**	**0.94**	**0.98**	**1.58**	**1.88**	**1.61**	**1.65**

Figure 15:Spark's Speedup in comparison to Hadoop on running TeraSort

Spark and Hadoop have same performance when the input is small sized (1 to 8 Gb).However Spark executes faster when the input is larger than 320 million records. The advantage is more obvious with large input size. The

maximum speedup is observed when the input is 640 million records which is 1.88x faster than Hadoop.

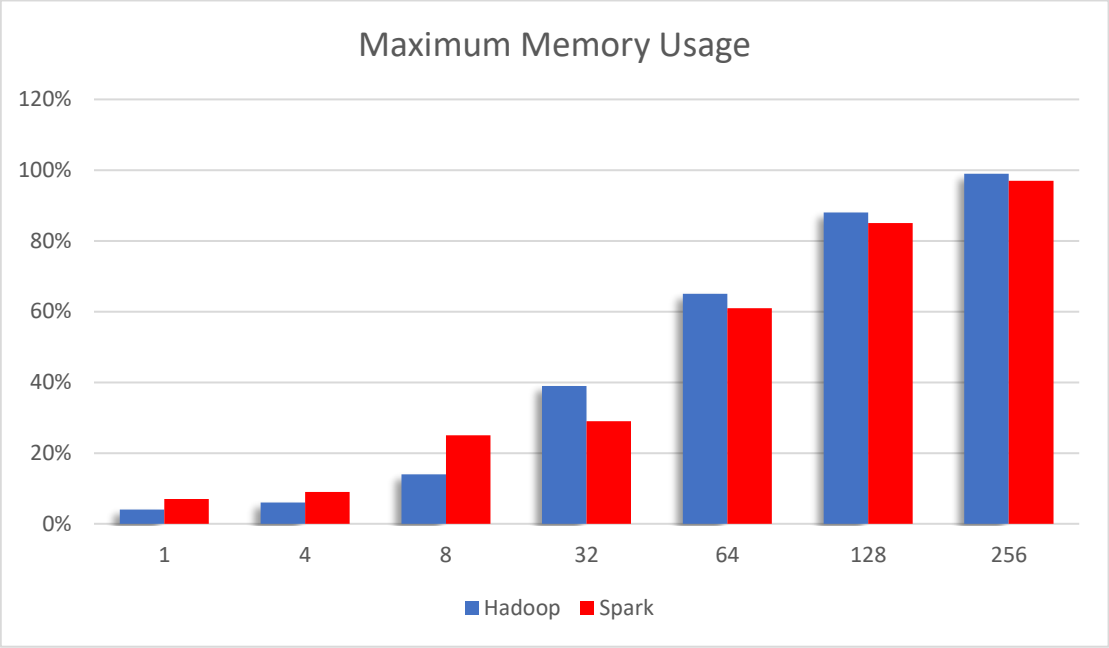


Figure 16:Maximum Memory Usage Comparison for running TeraSort on Hadoop and Spark

As shown in Figures 16 and 17, both systems have roughly same memory utilization. Spark is by far more CPU efficient (Figure 19). Hadoop has 4x more CPU consumption than Spark with input size 256 GB. Spark also has high throughput when the input is larger than 8 Gb as shown in Figure 18. Spark gives the maximum throughput with 100 MB per second when the input equals 320MB (Figure 18)

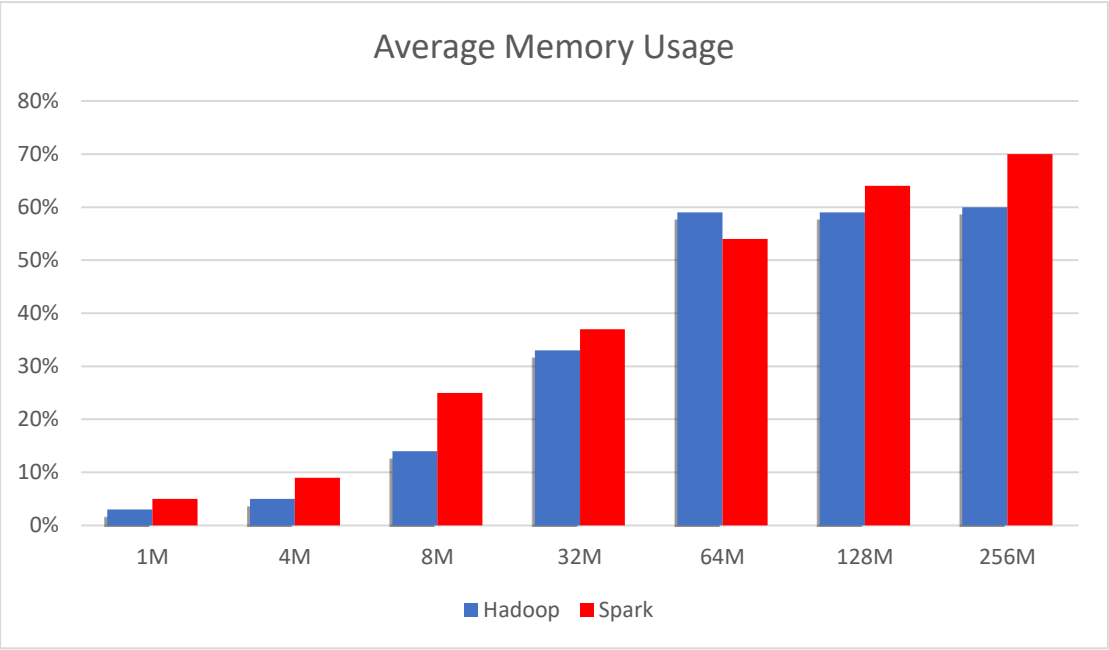


Figure 17:Average Memory Usage Comparison for running TeraSort on Hadoop and Spark

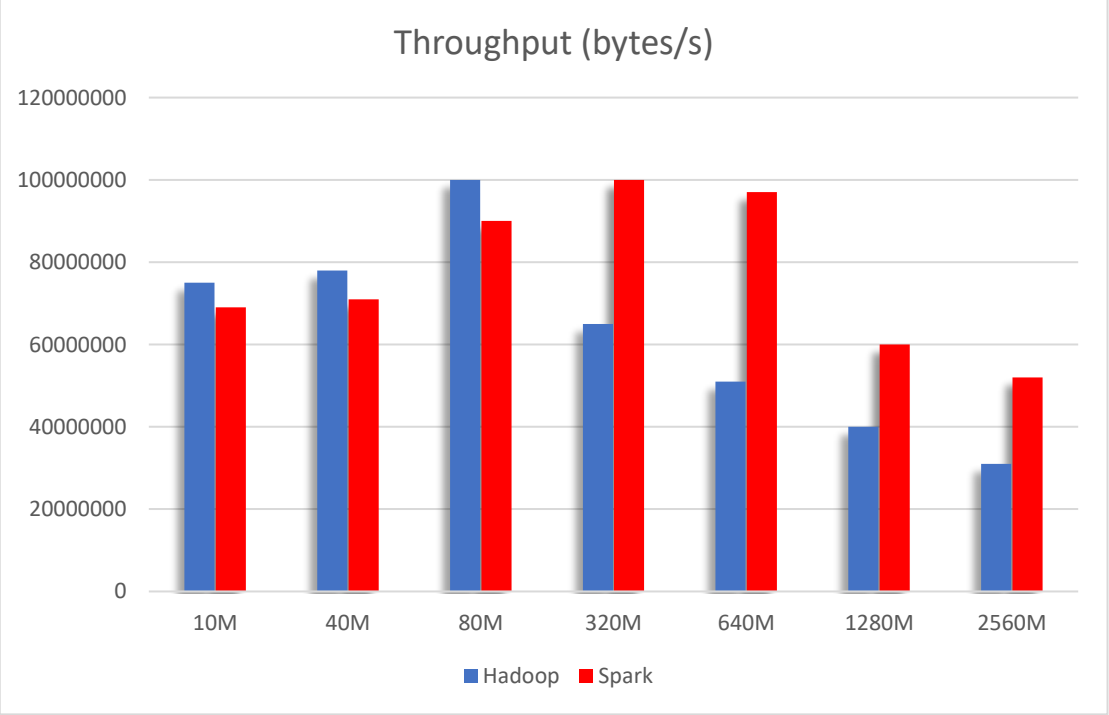


Figure 18: Throughput Comparison for running TeraSort on Hadoop and Spark

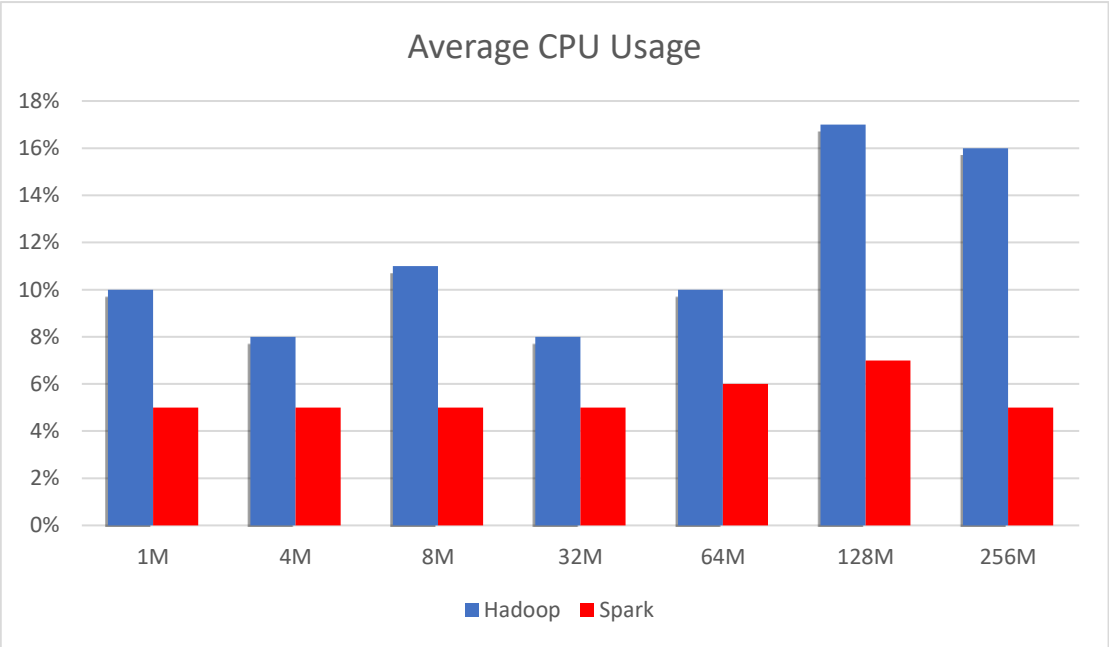


Figure 19: Average CPU Usage Comparison for running TeraSort on Hadoop and Spark

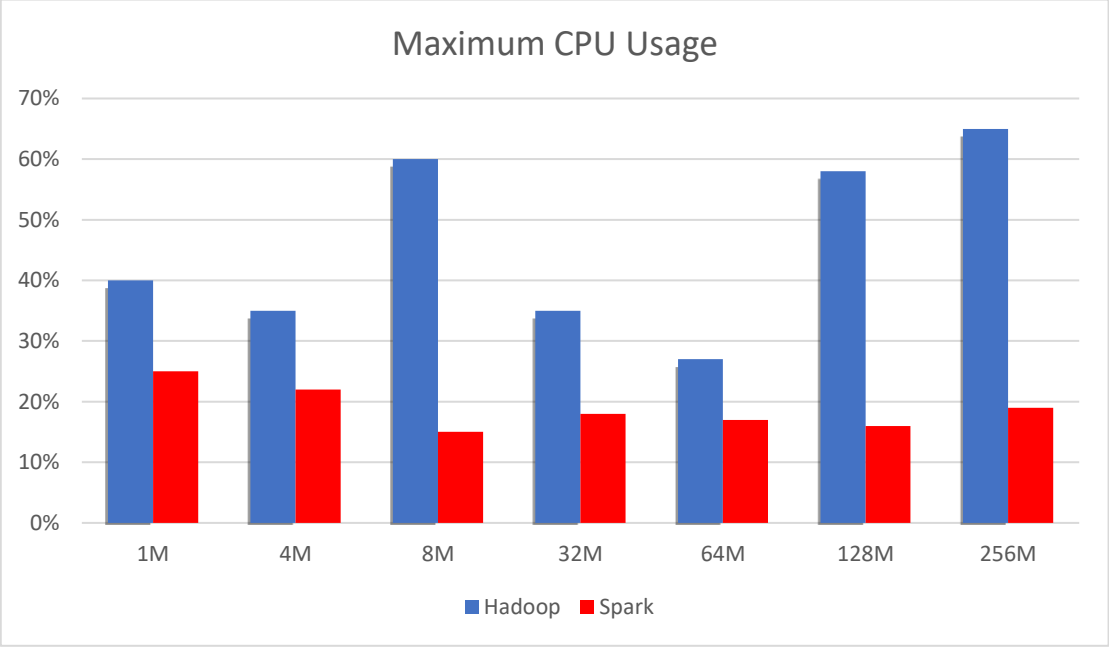


Figure 20:Maximum CPU Usage Comparison for running TeraSort on Hadoop and Spark

2.1.2 Naive Bayes

Naive Bayes classifiers is a perfect match of Machine Learning and big data as one of the simplest and fastest algorithms for classification [13]. The textbook application of Naive Bayes (NB) classifiers is spam filtering, where word frequency counts are used to classify whether a given message is spam or not. The Naive Bayes Classifier algorithm is based on Bayes' theorem with independence assumptions between the features to categorize text. There are two steps in Naive Bayes: training and testing. In the training step, the classifier is trained by the sample text file and get a model. In the testing step, the classifier processes the input data based on the model.

In this example, however, we're going to be using continuous data instead. More specifically, we'll be classifying flowers based on measurements of their petals size.

As with any classifier, the training data is a set of training examples x, each of which is composed of n features $x_i = (x_1 , x_2 , \dots , x_n)$ and their corresponding class C_i where i is one of k classes. The goal is to learn a conditional probability model:

$$p(C_k|x_1 , x_2 , \dots , x_k)$$

for each of the k classes in the dataset.Intuitively, learning this multivariate distribution will require a lot of data as the number of features grows. However, we can simplify the task if we assume that features are conditionally independent given the class. While this assumption never holds on real data, it results in a single but surprisingly simple classifier.

TensorFlow Implementation:

We start by grouping the training samples based on their labeled class and get a *(nb_classes * nb_samples * nb_features)* array.

Based on the above, we can fit individual Gaussian distributions to each combination of labeled class and feature. It's important to point out that, even if we're feeding the data in one go, we are fitting a series of univariate distributions, rather than a multivariate one:

```
mean, var = tf.nn.moments(tf.constant(points_by_class), axes=[1])
```

In this trivial example, we're using *tf.constant* to get the training data inside the TensorFlow graph. In real life, you probably want to use *tf.placeholder* or even more performing alternatives like *tf.Data*.We take advantage of TensorFlow's tf.distributions module to create a Gaussian distribution with the estimated mean and variance:

```
self.dist = tf.distributions.Normal(loc=mean, scale=tf.sqrt(var))
```

This distribution is the only thing we need to keep around for inference, and it's luckily pretty compact, since the mean and variance are only *(nb_classes, nb_features).*

For inference, it's important to work in the log probability space to avoid numerical errors due to repeated multiplication of small probabilities. We have:

$$\log p(C_k|x) = logp(C_k) + \sum_{i=1}^{n} P(x|C_k)$$

To take care of the first term, we can assume that all classes are equally likely (i.e. uniform prior):

```
priors = np.log (np.array([1. / nb_classes] * nb_classes))
```

To compute the sum in the second term, we duplicate (*tile*) the feature vectors along a new "class" dimension, so that we can get probabilities from the distribution in a single run:

```
# (nb_samples, nb_classes, nb_features)
all_log_probs = self.dist.lob_prob(tf.reshape(tf.tile(X, [1,nb_classes]), [-1, nb_classes, nb_features]))
```

The next step is to add up the contributions of each feature to the likelihood of each class. In TensorFlow lingo, this is a *reduce* operation over the features axis:

```
# (nb_samples, nb_classes)
cond_probs = tf.reduce_sum(all_log_probs, axis=2)
```

We can then add up the priors and the conditional probabilities to get the posterior distribution of the class label given the features:

```
joint_likelihood = tf.add(priors, cond_probs)
```

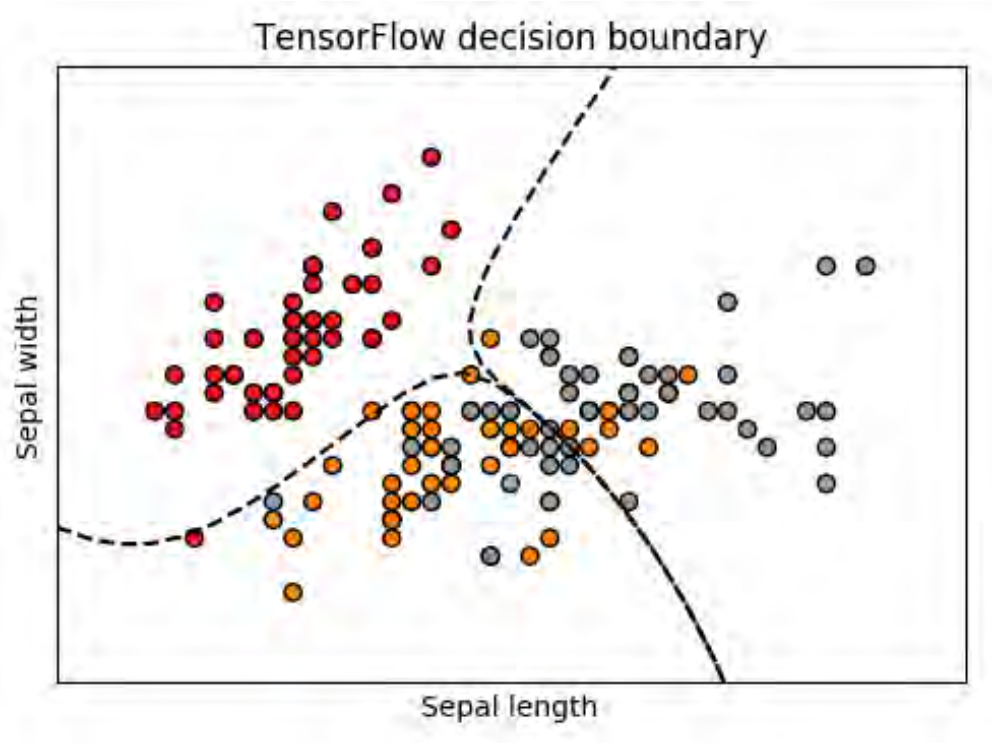
In the derivation, we ignored the normalization factor, so the expression above is not a proper probability distribution because it doesn't add up to 1. We fix that by subtracting a normalization factor in log space using TensorFlow's *reduce_logsumexp*. Naively computing *log(sum(exp(..)))*

```
    norm_factor = tf.reduce_logsumexp(
joint_likelihood, axis=1, keep_dims=True)                    log_prob =
joint_likelihood - norm_factor
```

Finally, we exponentiate to get actual probabilities:

```
    probs = tf.exp(log_prob)
```

By feeding in a grid of points and drawing the contour lines at 0.5 probability, we get a nice plot:



The following graph (figure 21) represents a running time performance comparison for Naïve Bayes between Hadoop and Spark.

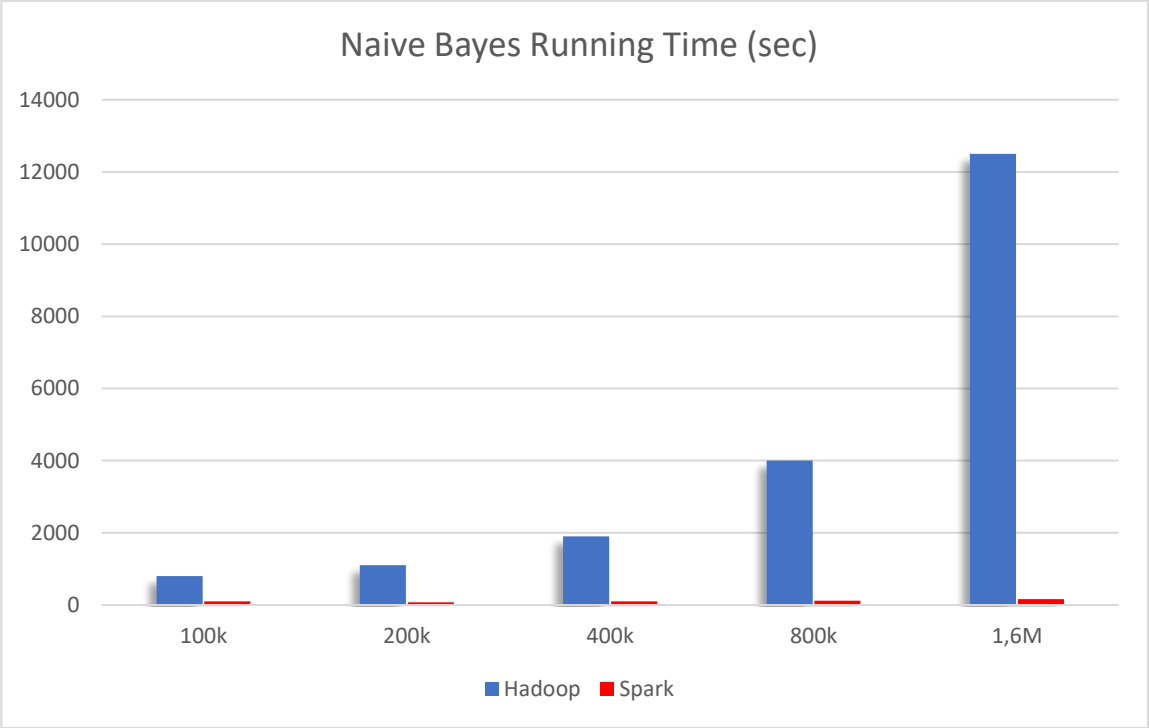


Figure 21:Running Time Comparison for Naive Bayes Running on Hadoop and Spark

Input	100k	200k	400k	800k	1.6M
Hadoop	541	917	1650	3537	10946
Spark	37	48	60	75	126
Speedup	**14.62**	**19.1**	**27.5**	**47.16**	**86.87**

Figure 22:Spark's Speedup over Hadoop on Running Naive Bayes

The input sets range from 100K to 1.6M and the input sizes range from 0.4 Gb to 7 Gb. Naive Bayes is a machine learning benchmark. Spark is designed for iterative jobs which reuse the same data set to optimize a parameter that it supposes to have a better performance than Hadoop on machine learning algorithms. As shown in Figure 21, Spark has a big

advantage on running Naïve Bayes workload especially with large input data sizes. As we can see in Figure 22, the speedup goes from 14.62x to 86.87x as the input changes. In addition, Spark has little improvements on average CPU utilization (Figure 26 and Figure 27) and concerning memory, both Spark and Hadoop use memory practically in the same way (Figure 23 and Figure 24). Spark has higher throughput than Hadoop and the throughput increases as the input size goes larger according to Figure 25.

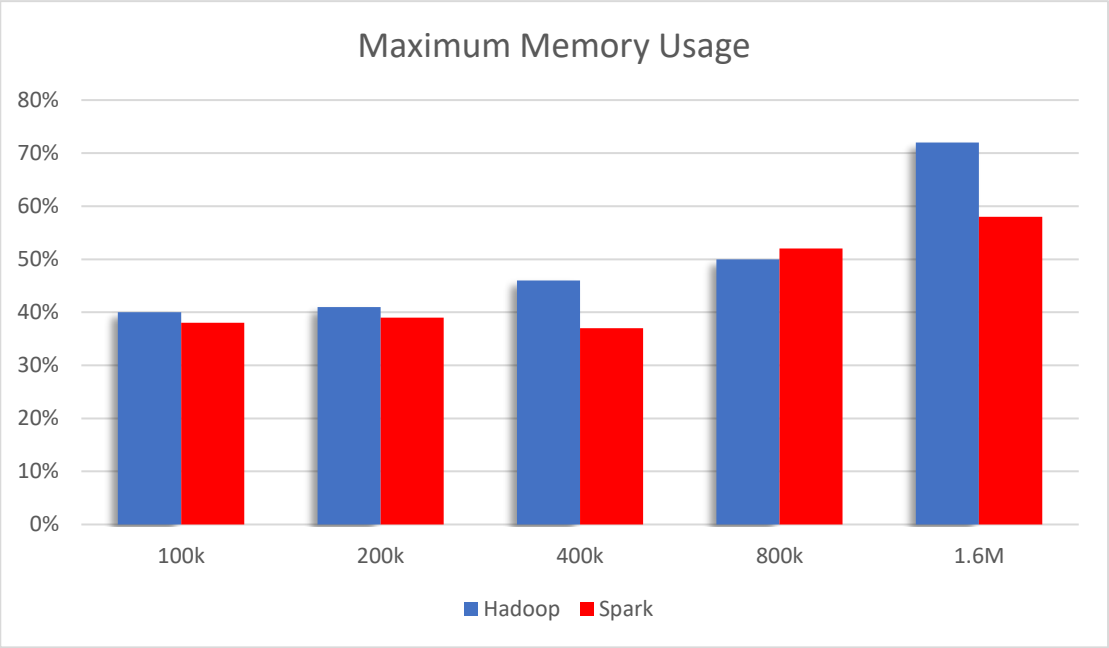


Figure 23:Maximum Memory Usage Percentage running Naive Bayes

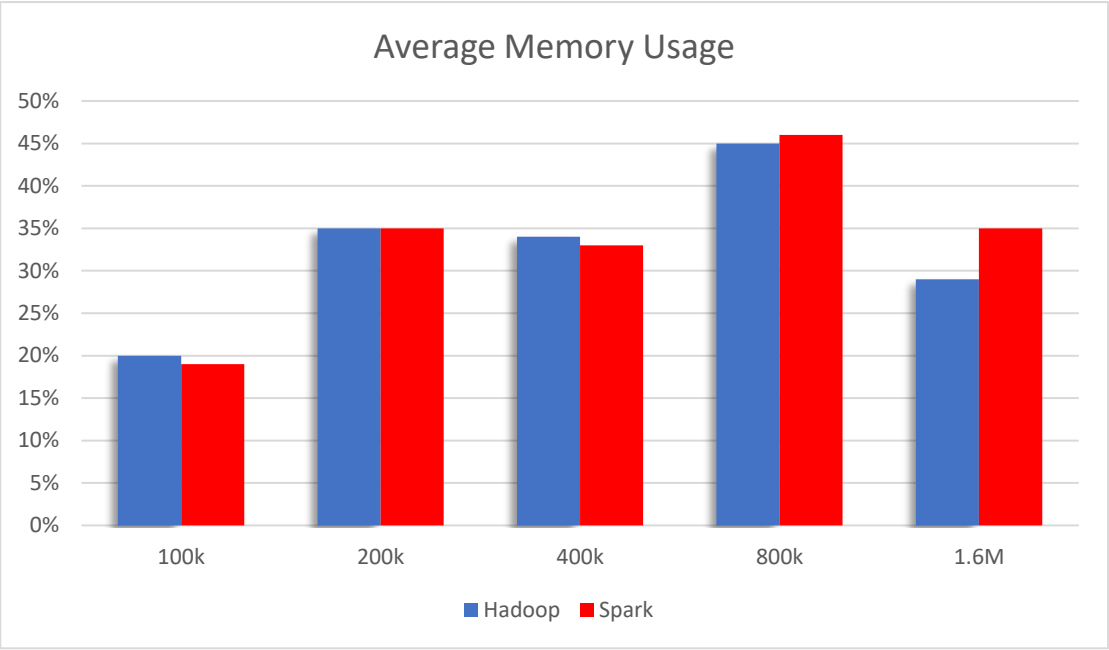


Figure 24:Average Memory Usage Percentage for Naive Bayes running on Hadoop and Spark

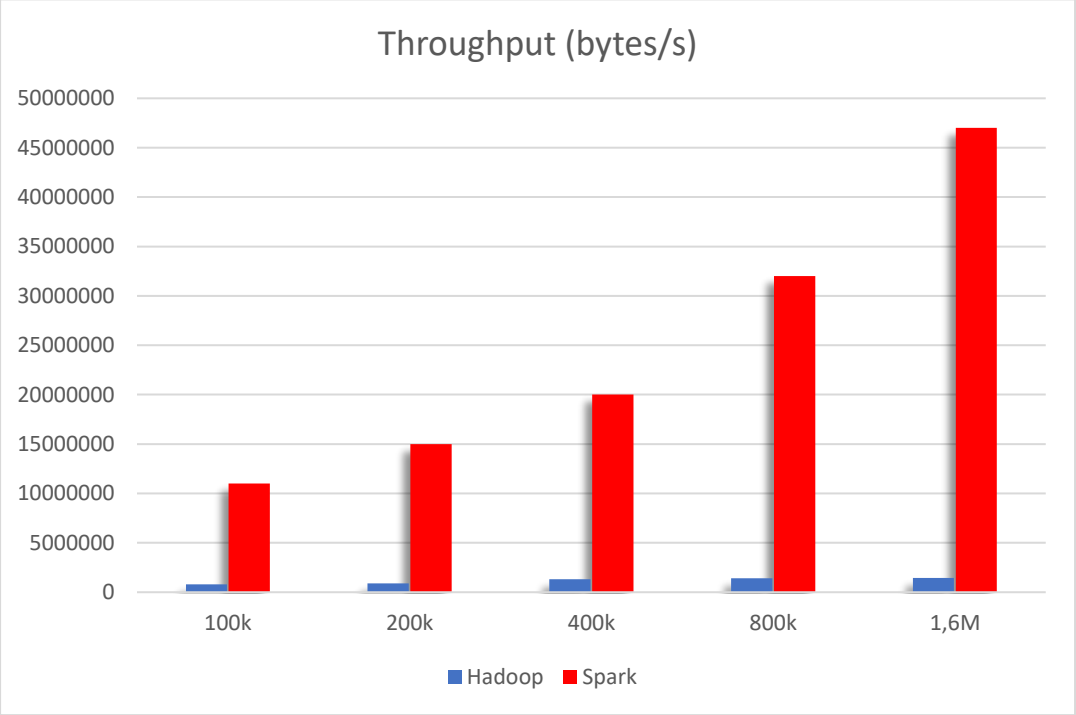


Figure 25:Throughput Comparison for Naive Bayes running on Hadoop and Spark

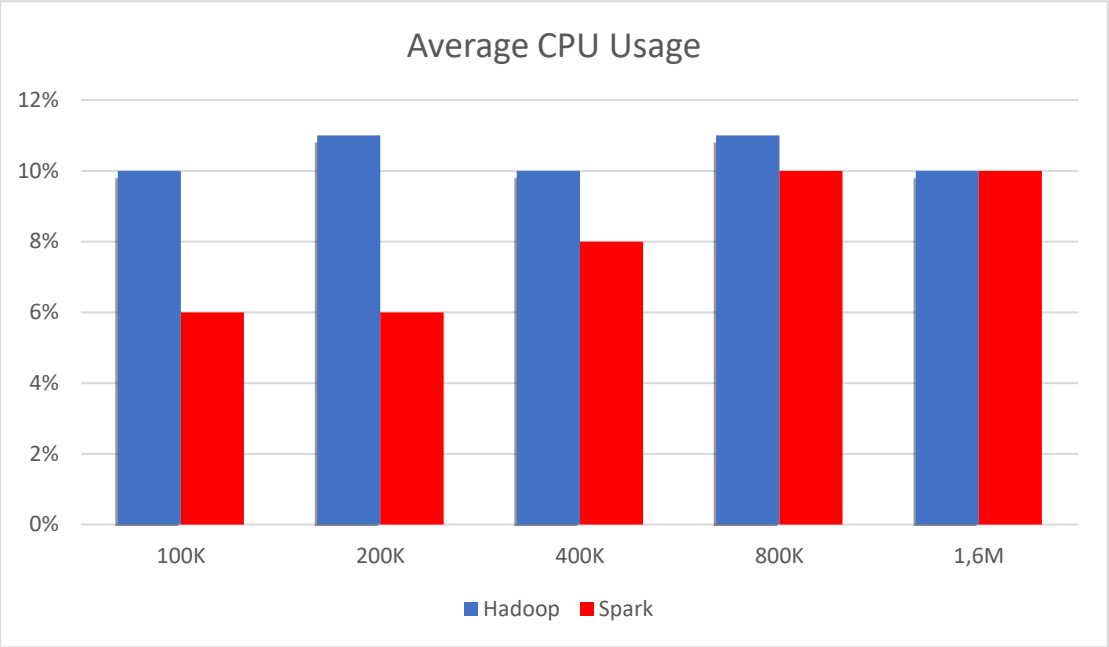


Figure 26:Average CPU Usage Percentage for Naive Bayes running on Hadoop and Spark

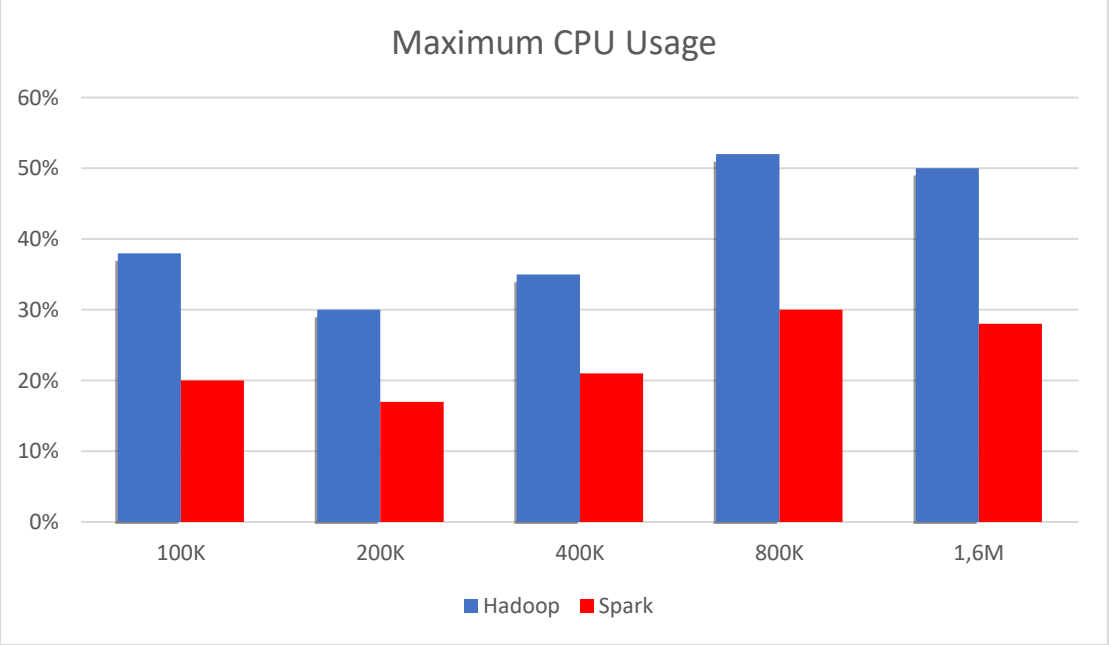


Figure 27:Maximum CPU Usage Percentage for Naive Bayes running on Hadoop and Spark

2.1.3 K-means

We implemented and used a TensorFlow K-means algorithm for grouping data into clusters with similar characteristics. When working with *k*-means, the data in a training set does not need labels. As an unsupervised learning method, the algorithm builds clusters based on the data itself.

First, we generated random data points with a uniform distribution and assign them to a 2D tensor constant. Then, we randomly chose initial centroids from the set of data points.

```
    points = tf.constant(np.random.uniform(0, 10, (points_n, 2)))
    centroids = tf.Variable(tf.slice(tf.random_shuffle(points), [0,
0], [clusters_n, -1]))
```

For the next step, we want to be able to do element-wise subtraction of points and centroids that are 2D tensors. Because the tensors have different shape, let's expand points and centroids into 3 dimensions, which allows us to use the broadcasting feature of subtraction operation.

```
    points_expanded = tf.expand_dims(points, 0)
    centroids_expanded = tf.expand_dims(centroids, 1)
```

Then, calculate the distances between points and centroids and determine the cluster assignments.

```
    distances = tf.reduce_sum(tf.square(tf.sub(points_expanded,
centroids_expanded)), 2)
    assignments = tf.argmin(distances, 0)
```

Next, we can compare each cluster with a cluster assignments vector, get points assigned to each cluster, and calculate mean values. These mean values are refined centroids, so let's update the centroids variable with the new values.

```
means = []
for c in xrange(clusters_n):
    means.append(tf.reduce_mean(
      tf.gather(points,
                tf.reshape(
                  tf.where(
                    tf.equal(assignments, c)
                  ),[1,-1])
                ),reduction_indices=[1]))
```

```
    new_centroids = tf.concat(0, means)
    update_centroids = tf.assign(centroids, new_centroids)
```

it’s time to run the graph. For each iteration, we update the centroids and return their values along with the cluster assignments values.

```
    with tf.Session() as sess:
      sess.run(init)
      for step in xrange(iteration_n):
        [_, centroid_values, points_values, assignment_values] =
sess.run([update_centroids, centroids, points, assignments])
```

Lastly, we display the coordinates of the final centroids and a multi-colored scatter plot showing how the data points have been clustered.

```
    print "centroids" + "\n", centroid_values

    plt.scatter(points_values[:, 0], points_values[:, 1],
c=assignment_values, s=50, alpha=0.5)
    plt.plot(centroid_values[:, 0], centroid_values[:, 1], 'kx',
markersize=15)
    plt.show()
```

The following graph (Figure 28)represents a running time performance comparison for K-means between Hadoop and Spark:

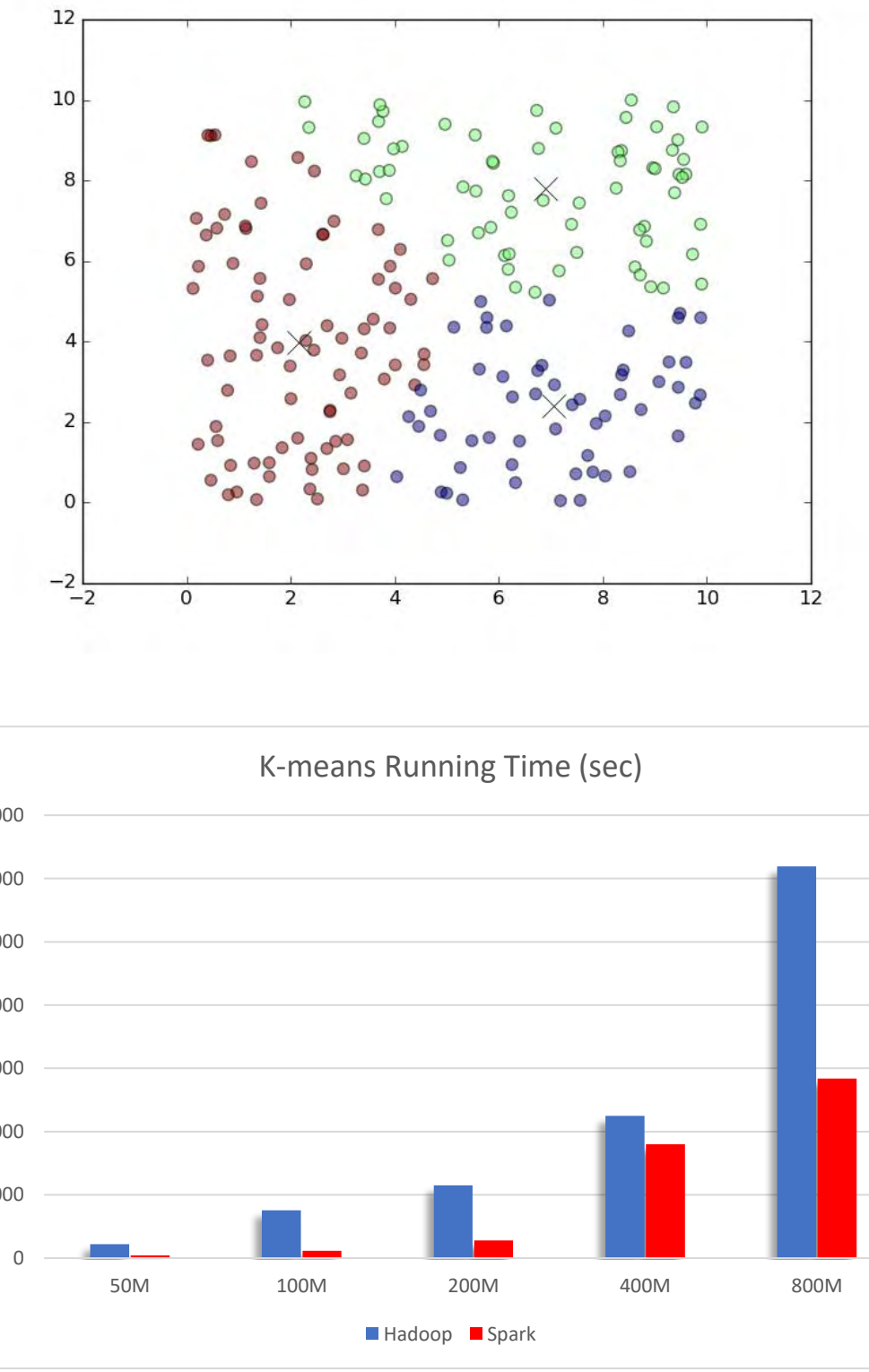


Figure 28:Running Time Comparison for K-means Running on Hadoop and Spark

Input	50M	100M	200M	400M	800M
Hadoop	220	758	1152	2250	6191
Spark	45	118	280	1802	2840
Speedup	**4.9**	**6.3**	**4.1**	**1.24**	**2.17**

Figure 29:Spark's Speedup over Hadoop on Running K-means

Figure 28 summarizes Spark's speedup over Hadoop on the same data sets. K-means is a machine learning algorithm which should be suitable for Spark: The input data points are assigned to clusters with a closest centroid and new centroids are created by these points assigned in the clusters. These steps are repeated until it converges. For each time, Hadoop need to store the intermediate results back to the disk. In contrast, Spark keeps them in Memory. The input sets range from 100K samples to 1.6M samples and the input sizes range from 10Gb to 160 Gb. Figure 29 shows that Spark has better performance than Hadoop -speedup is up to 6.3 times. However, the advantage is clearly bounded by the memory. The speedup goes down when the input is more than 100 million samples and has the minimum value 1.21x when the input is 400M. As shown in Figure 30 the maximum memory usage for Spark is almost 100 percent with 400M and 800M input Spark cannot create more RDD's at this point. Spark saves more CPU resources compared to Hadoop especially when talking for small data inputs. With inputs of 50M and 100M Spark's maximum CPU consumption is half of Hadoop's. When the input is smaller than 200 million samples, Spark shows improvements on throughput than Hadoop- up to 6.3x-. For inputs greater than 200 million samples, the throughput for Spark has an obvious decrement as seen by Figure 32

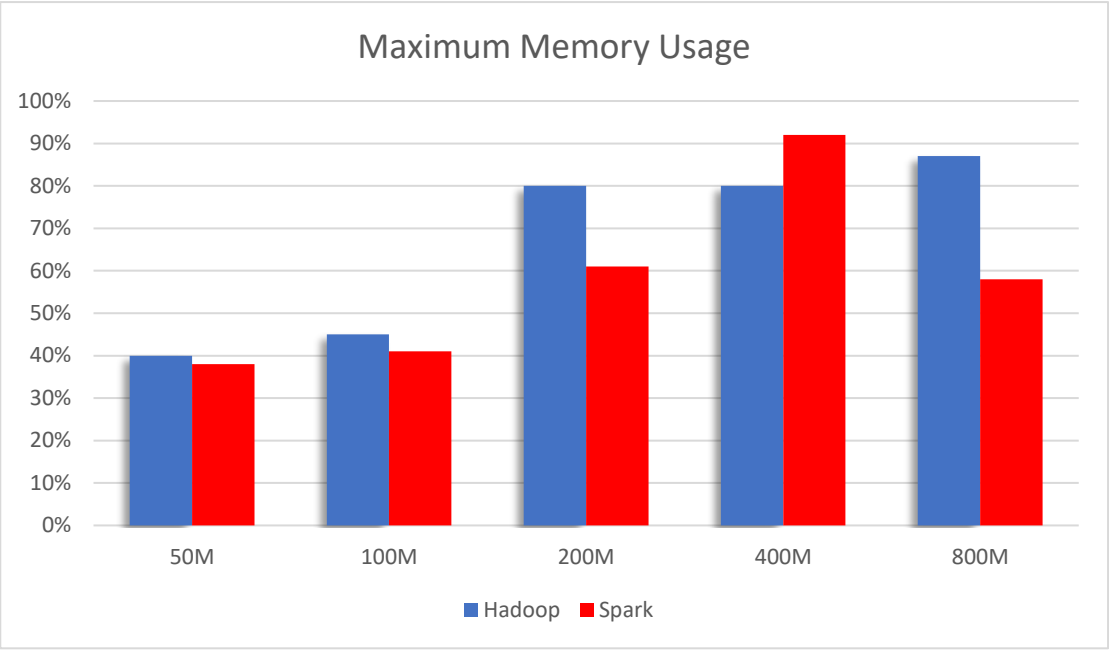


Figure 30:Maximum Memory Usage Percentage running K-means

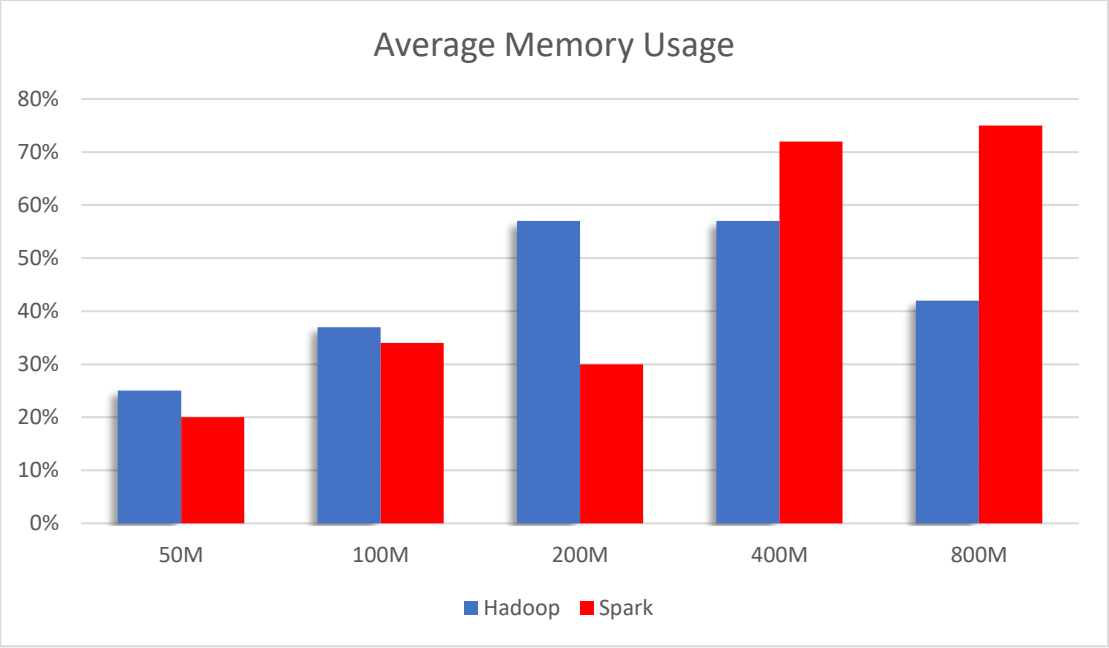


Figure 31:Average Memory Usage Percentage running K-means

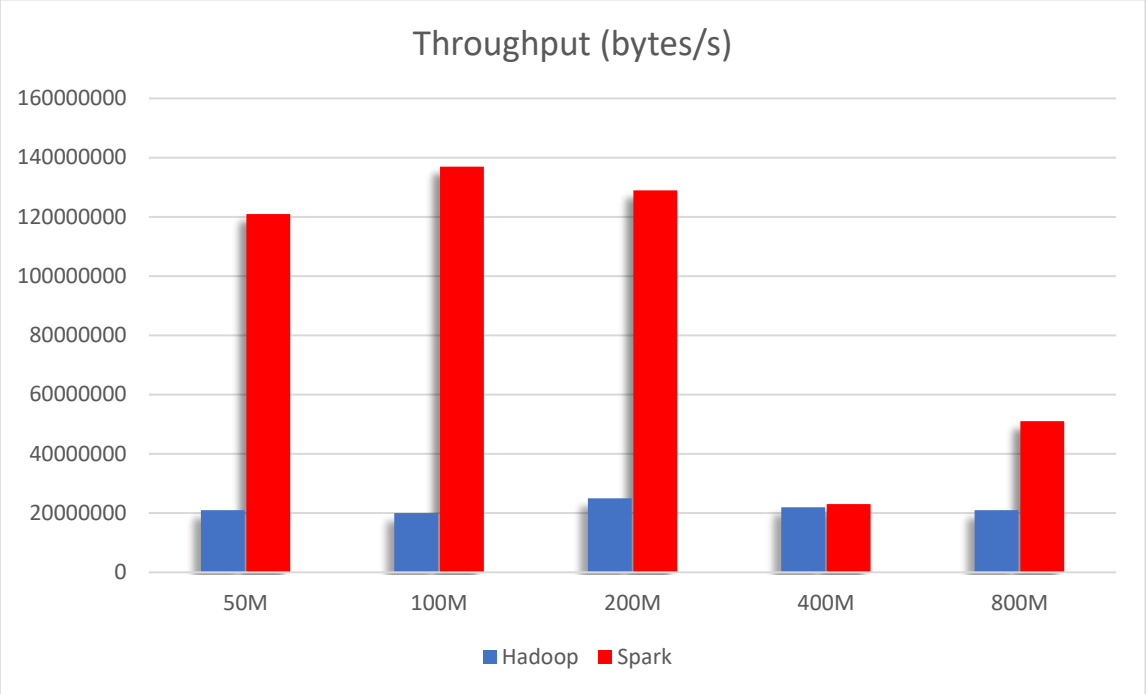


Figure 32:Throughput Comparison for k-MEANS running on Hadoop and Spark

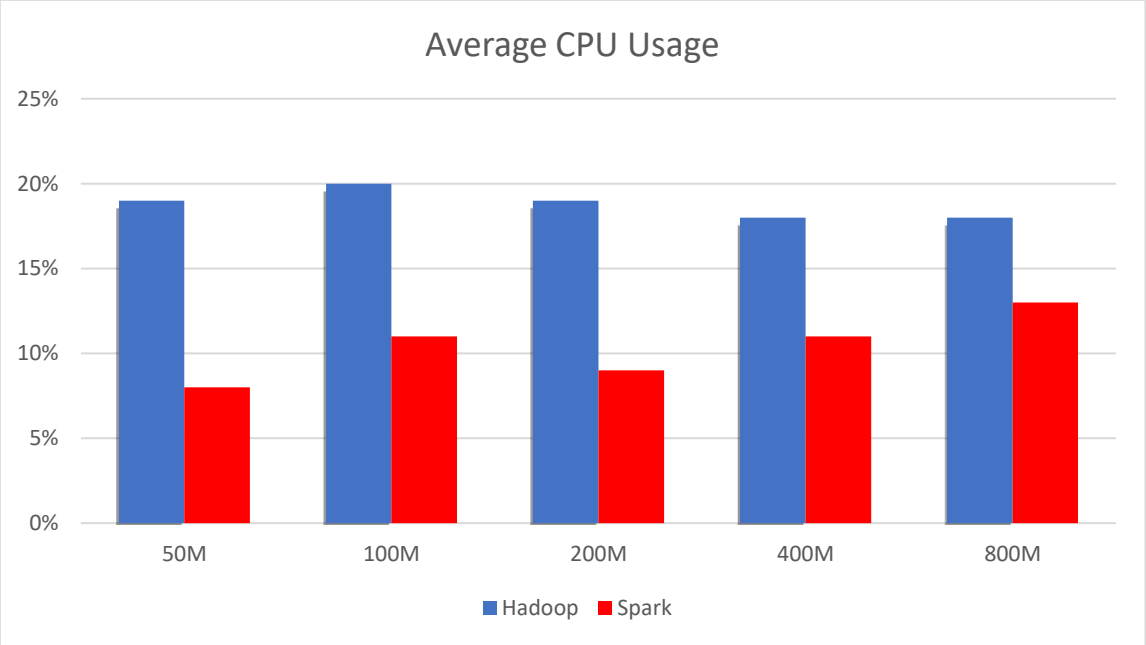


Figure 33: Average CPU Usage Percentage for K-means running on Hadoop and Spark

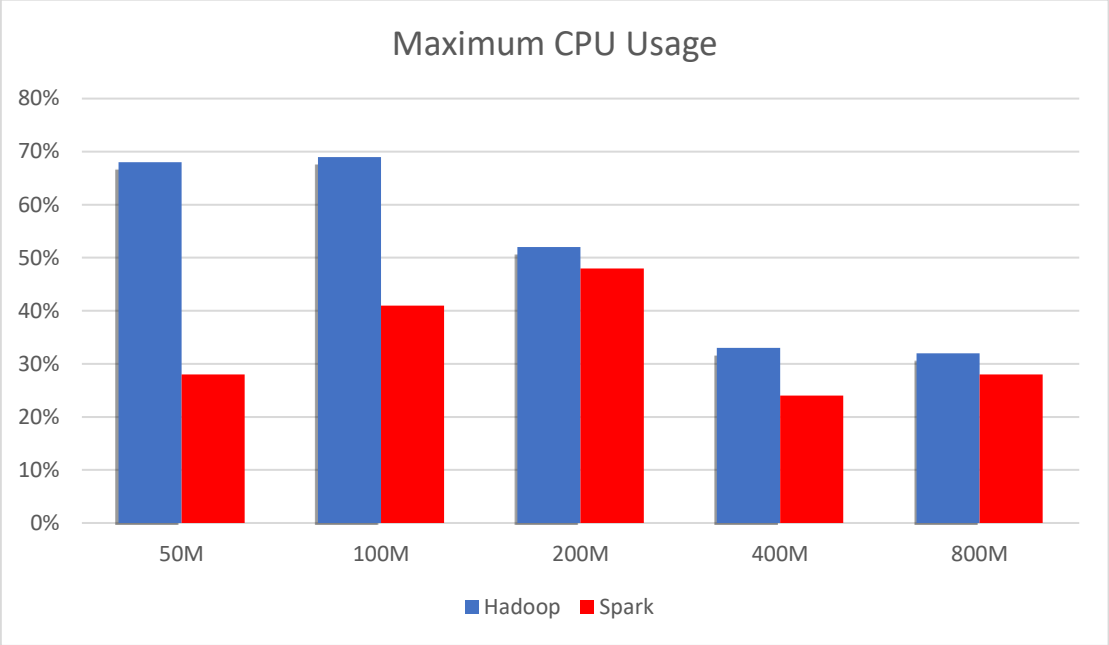


Figure 34:Maximum CPU Usage Percentage for K-means running on Hadoop and Spark

3. Verdicts analysis and future Work

From the benchmarking and evaluating work, we can conclude that Spark totally overshadows Hadoop on performance in all of case studies, especially those involved in iterative algorithms. We conclude that several factors can give a rise to a significant performance difference. First, Spark pipelines RDDs transformations and keeps persistent RDDs in memory by default, but Hadoop mainly concentrates on high throughput of data rather than on job execution performance such that MapReduce results in overheads due to data replication, disk I/O, and serialization, which can dominate application execution times. Also, in order to achieve fault-tolerance efficiently, RDDs provide a coarse-grained transformation rather than fine-grained updates to shared state or data replication across cluster [10], which means Spark builds the lineage of RDDs through transformations rather than the actual data. For example, if a partition of an RDD is missing, the RDD can retrieve the information about how it was originated from other RDDs. Last but not least, Spark has more optimizations, such as the number of disk accesses per second, memory bandwidth utilization and IPC rate, than Hadoop, so that it provides a better performance. Spark is generally faster than Hadoop because it is at the expense of significant memory consumption. But Spark is not a good fit for applications that make asynchronous fine-grained updates to shared state [10]. Also, if we do not have sufficient memory and the speed is not a demanding requirement, Hadoop is a better choice. For those applications which are time sensitive or involved in iterative algorithms and there is abundant memory available, Spark is sure to be the best fit.

As our future work, we plan to set up Hadoop and Spark on a bigger cluster to test the scalability of each platform. Also, we want to increase the memory capacity of the clusters and in order to explore the influence of memory restriction on running time of Spark we would like to use active fiber connection between nodes.

Finally, there is a big desire to design an intelligent system that can help us to choose a platform and the configuration parameters based on the applications and the input data sizes to get the optimized performance.

References

[0] Josh James. How much data is created every minute. https://www.domo.com/blog/2012/06/how-much-data-is-created-every-minute/

[1] *TensorFlow documentation*, https://www.tensorflow.org/

[2] Itay Lieder, Tom Hope, and Yehezkel S. Resheff: *Learning TensorFlow: A Guide to Building Deep Learning Systems*, CA: O'Reilly Media, 2017

[3] *Apache Hadoop documentation*, https://hadoop.apache.org/

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung Google, ” The Google File System”, *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, ACM, Bolton Landing, NY (2003), pp. 20-43

[5] Dean, Jeffrey; Ghemawat, Sanjay, MapReduce: “Simplified Data Processing on Large Clusters”*, OSDI'04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA (2004)*, pp. 137-150

[6] T. White, Hadoop: *The Definitive Guide* (Fourth edition). Sebastopol, CA: O'Reilly Media, 2015.

[7] *Spark Overview,* 2018, http://spark.apache.org/.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: a fault-tolerant abstraction for inmemory cluster computing,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (NSDI'12), USENIX Association, Berkeley,2012, p2

[9] Holden Karau and Rachel Warren : *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark*, CA: O'Reilly Media, 2017

[10] N. Islam, S. Sharmin, M. Wasi-ur-Rahman, X. Lu, D. Shankar, D. K. Panda, "*Performance characterization and acceleration of in-memory file systems for Hadoop and Spark applications on HPC clusters*," in *2015 IEEE International Conference on Big Data (Big Data),* October 29, 2015-November 1, 2015, pp. 243-252.

[11] *Yarn overview and documentation*, https://yarnpkg.com/lang/en/docs/

[12] Lei Gu and Huan Li. Memory or time: *"Performance evaluation for iterative operation on hadoop and spark."* ,IEEE International Conference, 2013.

[13*]* *Naïve Bayes for Machine Learning article* : https://towardsdatascience.com/naive-bayes-in-machine-learning-/

[14] Satish Gopalani and Rohan Arora*."Comparing apache spark and map reduce with performance analysis using k-means "*. International Journal of Computer Applications, 113(1), March 2015.

[15] *TeraSort benchmark* , https://mapr.com/resources/terasort-benchmark-comparison-yarn/